XForms

XForms (Forms Library)
A Graphical User Interface Toolkit for X

Library Version 1.2

January 2014

Table of Contents

Preface 1

Part I - Using the Forms Library 6

1 Introduction 7

2 Getting Started 9
2.1 Naming Conventions 9
2.2 Some Examples 9
2.3 Programming Model 13

3 Defining Forms 15
3.1 Starting and Ending a Form Definition 15
3.2 Boxes 15
3.3 Texts 17
3.4 Buttons 18
3.5 Sliders 19
3.6 ValSliders 20
3.7 Input Fields 20
3.8 Grouping Objects 21
3.9 Hiding and Showing 22
3.10 Deactivating and Triggering Objects 22
3.11 Changing Attributes 23
3.11.1 Color 23
3.11.2 Bounding Boxes 26
3.11.3 Label Attributes and Fonts 27
3.11.4 Tool Tips 31
3.11.5 Redrawing Objects 32
3.11.6 Changing Many Attributes 32
3.11.7 Symbols 32
3.12 Adding and Removing Objects 35
3.13 Freeing Objects 35

4 Doing Interaction 37
4.1 Displaying a Form 37
4.2 Simple Interaction 43
4.3 Periodic Events and Non-blocking Interaction 46
4.4 Dealing With Multiple Windows 48
4.5 Using Callback Functions 51
4.6 Handling Other Input Sources 54

5 Free Objects . . . 56
5.1 Free Object . . . 56
5.2 An Example . . . 59

6 Goodies . . . 69
6.1 Messages and Questions . . . 69
6.2 Command Log . . . 73
6.3 Colormap . . . 74
6.4 File Selector . . . 75

***Part II - The Form Designer* . . . 81**

7 Introduction . . . 82

8 Getting Started . . . 83

9 Command Line Arguments . . . 86

10 Creating Forms . . . 88
10.1 Creating and Changing Forms . . . 88
10.2 Adding Objects . . . 88
10.3 Selecting Objects . . . 88
10.4 Moving and Scaling . . . 89
10.5 Aligning Objects . . . 89
10.6 Raising and Lowering . . . 90
10.7 Setting Attributes . . . 90
10.8 Generic Attributes . . . 91
10.8.1 Basic Attributes . . . 91
10.8.2 Font . . . 92
10.8.3 Misc. Attributes . . . 92
10.8.4 Colors . . . 92
10.9 Object Specific Attributes . . . 93
10.10 Cut, Copy and Paste . . . 93
10.11 Groups . . . 94
10.12 Hiding and Showing Objects . . . 94
10.13 Testing Forms . . . 94

11 Saving and Loading Forms . . . 96

12 Language Filters . . . 102
12.1 External Filters . . . 102
12.2 Command Line Arguments of the Filter . . . 103

13 Generating Hardcopies . . . 104

Part III - Object Classes 106

14 Introduction 107

15 Static Objects 109
15.1 Box Object 109
15.1.1 Adding Box Objects 109
15.1.2 Box Types 109
15.1.3 Box Attributes 109
15.1.4 Remarks 110
15.2 Frame Object 110
15.2.1 Adding Frame Objects 110
15.2.2 Frame Types 110
15.2.3 Frame Attributes 111
15.2.4 Remarks 111
15.3 LabelFrame Object 111
15.3.1 Adding LabelFrame Objects 111
15.3.2 LabelFrame Types 111
15.3.3 Attributes 112
15.3.4 Remarks 112
15.4 Text Object 112
15.4.1 Adding Text Objects 112
15.4.2 Text Types 112
15.4.3 Text Attributes 112
15.4.4 Remarks 113
15.5 Bitmap Object 113
15.5.1 Adding Bitmap Objects 113
15.5.2 Bitmap Types 113
15.5.3 Bitmap Interaction 113
15.5.4 Other Bitmap Routines 113
15.5.5 Bitmap Attributes 114
15.5.6 Remarks 114
15.6 Pixmap Object 114
15.6.1 Adding Pixmap Objects 114
15.6.2 Pixmap Types 114
15.6.3 Pixmap Interaction 114
15.6.4 Other Pixmap Routines 114
15.6.5 Pixmap Attributes 115
15.6.6 Remarks 115
15.7 Clock Object 116
15.7.1 Adding Clock Objects 116
15.7.2 Clock Types 116
15.7.3 Clock Interaction 116
15.7.4 Other Clock Routines 117
15.7.5 Clock Attributes 117
15.7.6 Remarks 117
15.8 Chart Object 117

15.8.1 Adding Chart Objects . . . 117
15.8.2 Chart Types . . . 117
15.8.3 Chart Interaction . . . 118
15.8.4 Other Chart Routines . . . 118
15.8.5 Chart Attributes . . . 119
15.8.6 Remarks . . . 119

16 Button-like Objects . . . 120

16.1 Adding Button Objects . . . 120
16.2 Button Types . . . 122
16.3 Button Interaction . . . 123
16.4 Other Button Routines . . . 124
16.5 Button Attributes . . . 126
16.6 Remarks . . . 126

17 Valuator Objects . . . 127

17.1 Slider Object . . . 127
17.1.1 Adding Slider Objects . . . 127
17.1.2 Slider Types . . . 127
17.1.3 Slider Interaction . . . 128
17.1.4 Other Slider Routines . . . 129
17.1.5 Slider Attributes . . . 129
17.1.6 Remarks . . . 130
17.2 Scrollbar Object . . . 131
17.2.1 Adding Scrollbar Objects . . . 131
17.2.2 Scrollbar Types . . . 131
17.2.3 Scrollbar Interaction . . . 132
17.2.4 Other Scrollbar Routines . . . 133
17.2.5 Scrollbar Attributes . . . 133
17.2.6 Remarks . . . 134
17.3 Dial Object . . . 134
17.3.1 Adding Dial Objects . . . 134
17.3.2 Dial Types . . . 134
17.3.3 Dial Interaction . . . 134
17.3.4 Other Dial Routines . . . 135
17.3.5 Dial Attributes . . . 136
17.3.6 Remarks . . . 136
17.4 Positioner Object . . . 136
17.4.1 Adding Positioner Objects . . . 136
17.4.2 Positioner Types . . . 136
17.4.3 Positioner Interaction . . . 136
17.4.4 Other Positioner Routines . . . 137
17.4.5 Positioner Attributes . . . 138
17.4.6 Remarks . . . 138
17.5 Counter Object . . . 138
17.5.1 Adding Counter Objects . . . 139
17.5.2 Counter Types . . . 139

17.5.3 Counter Interaction . . . 139
17.5.4 Other Counter Routines . . . 140
17.5.5 Counter Attributes . . . 141
17.5.6 Remarks . . . 141
17.6 Spinner Object . . . 141
17.6.1 Adding Spinner Objects . . . 142
17.6.2 Spinner Types . . . 142
17.6.3 Spinner Interaction . . . 142
17.6.4 Other Spinner Routines . . . 143
17.6.5 Spinner Attributes . . . 143
17.7 Thumbwheel Object . . . 144
17.7.1 Adding Thumbwheel Objects . . . 144
17.7.2 Thumbwheel Types . . . 144
17.7.3 Thumbwheel Interaction . . . 144
17.7.4 Other Thumbwheel Routines . . . 145
17.7.5 Thumbwheel Attributes . . . 146
17.7.6 Remarks . . . 146

18 Input Objects . . . 147

18.1 Adding Input Objects . . . 147
18.2 Input Types . . . 147
18.3 Input Interaction . . . 148
18.4 Other Input Routines . . . 152
18.5 Input Attributes . . . 154
18.6 Remarks . . . 156

19 Choice Objects . . . 157

19.1 Select Object . . . 157
19.1.1 Adding Select Objects . . . 157
19.1.2 Select Interaction . . . 159
19.1.3 Other Select Routines . . . 160
19.1.4 Select Attributes . . . 162
19.1.5 Remarks . . . 163
19.2 Nmenu Object . . . 163
19.2.1 Adding Nmenu Objects . . . 163
19.2.2 Nmenu Interaction . . . 165
19.2.3 Other Nmenu Routines . . . 166
19.2.4 Nmenu Attributes . . . 168
19.2.5 Remarks . . . 168
19.3 Browser Object . . . 168
19.3.1 Adding Browser Objects . . . 169
19.3.2 Browser Types . . . 169
19.3.3 Browser Interaction . . . 169
19.3.4 Other Browser Routines . . . 171
19.3.5 Browser Attributes . . . 174
19.3.6 Remarks . . . 177

20 Container Objects . . . 178
20.1 Folder Object . . . 178
20.1.1 Adding Folder Objects . . . 178
20.1.2 Folder Types . . . 178
20.1.3 Folder Interaction . . . 178
20.1.4 Other Folder Routines . . . 179
20.1.5 Remarks . . . 181
20.2 FormBrowser Object . . . 182
20.2.1 Adding FormBrowser Objects . . . 182
20.2.2 FormBrowser Types . . . 182
20.2.3 FormBrowser Interaction . . . 182
20.2.4 Other FormBrowser Routines . . . 183
20.2.5 Remarks . . . 185

21 Other Objects . . . 186
21.1 Timer Object . . . 186
21.1.1 Adding Timer Objects . . . 186
21.1.2 Timer Types . . . 186
21.1.3 Timer Interaction . . . 186
21.1.4 Other Timer Routines . . . 186
21.1.5 Timer Attributes . . . 187
21.1.6 Remarks . . . 187
21.2 XYPlot Object . . . 187
21.2.1 Adding XYPlot Objects . . . 188
21.2.2 XYPlot Types . . . 188
21.2.3 XYPlot Interaction . . . 189
21.2.4 Other XYPlot Routines . . . 190
21.2.5 XYPlot Attributes . . . 196
21.2.6 Remarks . . . 196
21.3 Canvas Object . . . 197
21.3.1 Adding Canvas Objects . . . 197
21.3.2 Canvas Types . . . 197
21.3.3 Canvas Interaction . . . 197
21.3.4 Other Canvas Routines . . . 199
21.3.5 Canvas Attributes . . . 200
21.3.6 OpenGL Canvas . . . 200

22 Popups . . . 202
22.1 Adding Popups . . . 202
22.2 Popup Interaction . . . 211
22.3 Other Popup Routines . . . 213
22.4 Popup Attributes . . . 215

23 Deprecated Objects . . . 219
23.1 Choice Object . . . 219
23.1.1 Adding Choice Objects . . . 219
23.1.2 Choice Types . . . 219
23.1.3 Choice Interaction . . . 219
23.1.4 Other Choice Routines . . . 220
23.1.5 Choice Attributes . . . 221
23.1.6 Remarks . . . 222
23.2 Menu Object . . . 222
23.2.1 Adding Menu Objects . . . 222
23.2.2 Menu Types . . . 222
23.2.3 Menu Interaction . . . 223
23.2.4 Other Menu Routines . . . 223
23.2.5 Menu Attributes . . . 227
23.2.6 Remarks . . . 227
23.3 XPopup . . . 227
23.3.1 Creating XPopups . . . 227
23.3.2 XPopup Interaction . . . 232
23.3.3 Other XPopup Routines . . . 234
23.3.4 XPopup Attributes . . . 235
23.3.5 Remarks . . . 236

***Part IV - Designing Object Classes* . . . 237**

24 Introduction . . . 238

25 Global Structure . . . 239
25.1 The Routine `fl_add_NEW()` . . . 240

26 Events . . . 242
26.1 Shortcuts . . . 245

27 The Type `FL_OBJECT` . . . 247

28 Drawing Objects . . . 254
28.1 General Remarks . . . 254
28.2 Color Handling . . . 255
28.3 Mouse Handling . . . 256
28.4 Clipping . . . 257
28.5 Getting the Size . . . 258
28.6 Font Handling . . . 259
28.7 Drawing Functions . . . 259

29 An Example . . . 265

30 New Buttons **268**

31 Using a Pre-emptive Handler **275**

Part V - General Informations **276**

32 Overview of Main Functions **277**

32.1 Version Information 277
32.2 Initialization 277
32.3 Creating Forms 285
32.4 Object Attributes 286
32.5 Doing Interaction 292
32.6 Signals 298
32.7 Idle Callbacks and Timeouts 299
32.8 Global Variables 300

33 Some Useful Functions **302**

33.1 Misc. Functions 302
33.2 Windowing Support 302
33.3 Cursors 306
33.4 Clipboard 307

34 Resources for Forms Library **309**

34.1 Current Support 309
34.1.1 Resources Example 311
34.2 Going Further 312

35 Dirty Tricks **314**

35.1 Interaction 314
35.1.1 Form Events 314
35.1.2 Object Events 315
35.2 Other 315

36 Trouble Shooting **317**

Part VI - Image Support API **318**

37 Images . . . 319
37.1 The Basic Image Support API . . . 319
37.2 The FL_IMAGE Structure . . . 322
37.3 Supported image types . . . 325
37.4 Creating Images . . . 326
37.5 Supported Image Formats . . . 328
37.5.1 Built-in support . . . 328
37.5.2 Adding New Formats . . . 330
37.5.3 Queries . . . 335
37.6 Setup and Configuration . . . 336
37.7 Simple Image Processing . . . 338
37.7.1 Convolution . . . 338
37.7.2 Tint . . . 339
37.7.3 Rotation . . . 339
37.7.4 Image Flipping . . . 340
37.7.5 Cropping . . . 340
37.7.6 Scaling . . . 341
37.7.7 Warping . . . 342
37.7.8 General Pixel Transformation . . . 343
37.7.9 Image Annotation . . . 344
37.7.9.1 Using Text Strings . . . 344
37.7.9.2 Using Markers . . . 345
37.7.9.3 Pixelizing the Annotation . . . 347
37.7.10 Write Your Own Routines . . . 347
37.8 Utilities . . . 347
37.8.1 Memory Allocation . . . 347
37.8.2 Color Quantization . . . 348
37.8.3 Remarks . . . 349

Index of Functions . . . 350

Index of Global Variables . . . 359

Index of Constants . . . 360

Preface

The Forms Library for the X Window system (or XForms for short) is a GUI toolkit with a rather long history. It was developed in the last decade of the last millenium by **Dr. T. C. Zhao** (then at the Department of Physics, University of Wisconsin-Milwaukee, USA) and **Prof. Dr. Mark Overmars** (Department of Computer Science, Utrecht University, Netherlands) at a time when there were hardly any alternatives except expensive packages. While at first being closed source it became open source software in 2002, distributed according to the Lesser GNU Public License (LGPLv2).

While development slowed down a bit while other toolkits became available and matured, XForms is still used, and development continues. While it may not be as polished as newer toolkits it has the advantage of being relatively small and thus easier to get started with it.

The XForms home page is at

`http://xforms-toolkit.org/`

The sources and mailing list are hosted on

`https://savannah.nongnu.org/projects/xforms/`

The source package can be downloaded from

`http://download.savannah.gnu.org/releases/xforms/`

while the `git` repository can be accessed via

`git://git.savannah.nongnu.org/xforms.git`

`http://git.savannah.gnu.org/cgit/xforms.git`

`ssh://git.sv.gnu.org/srv/git/xforms.git`

There also is a mailing list. You can subscribe to it at

`http://lists.nongnu.org/mailman/listinfo/xforms-development`

The archive of the mailing list can be found at

`http://lists.gnu.org/archive/html/xforms-development/`

The archive of messages from before August 2009 and going back until 1996 is at

`http://xforms-toolkit.org/old-archive`

Please write to the mailing list if you have questions or find bugs.

This document is based on the documentation for version 0.89 of the Forms Library. It has been reconstructed from the PDF version (the original sources seem to have been lost) and has been updated to cover all changes introduced since version 0.89.

In the following the preface for the last available version of the documentation (version 0.89 from June 2000) is reproduced. Please note that quite a bit of the information there-in is outdated. Many of the URLs mentioned don't exist anymore, email addresses have changed and the restrictions on the distribution of the library have been removed by the original authors in favor of the LGPL.

Preface of Version 0.89 (June 2000)

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 extendible.

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, Linux(i386, alpha, m68k and sparc) as well as FreeBSD, NetBSD (i386, m68k and sparc), OpenBSD(i386, pmax, sparc, alpha), SCO and Unixware. Due to access and knowledge, testing on non-unix platforms such as OpenVMS, OS/2 and Microsoft/NT are less than comprehensive.

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 haveForm 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

Forms Library for X,

Forms Library

or simply

XForms

Forms Library is not public domain. It is copyright (c) by T.C. Zhao and Mark Overmars, and others, with all published and unpublished rights reserved. However, permission to use for non-commercial and not-for-profit purposes is granted. You may not use xforms commercially (including in-house and contract/consulting use) without contacting (xforms@world.std.com) for a license arrangement. Use of xforms for the sole purpose of running a publically available free software that requires it is not considered a commercial use, even in a commercial setting.

You may not "bundle" and distribute this software with commercial systems without prior consent of the authors. Permission to distribute this software with other free software that requires it, including Linux CD distribution, is granted. Further, permission to re-package the software is granted.

This software is provided "as is" without warranty of any kind, either expressed or implied. The entire risk as to the quality and performance of the software is with you. Should the software prove defective, you assume the cost of all necessary servicing, repair or correction and under no circumstance shall the authors be liable for any damages resulting from the use or mis-use of this software.

It would be appreciated if credit to the authors is acknowledged in published articles on applications based on the library. A reprint of the article would also be appreciated.

The development environment for xforms consists of Linux 1.0.8/a.out X11R5 and Linux 2.0/ELF X11R6 with additional testing and validation on SGI R8000 and occasionally IBM RS6000/AIX and other machines. For every public release, most of the demos and some internal testing programs are run on each platform to ensure quality of the distribution.

Figures in this document were produced by fd2ps, a program that takes the output of the form designer and converts the form definition into an encapsulated POSTSCRIPT file. fd2ps as ofXForms V0.85 is included in the distribution.

This document is dated June 12, 2000.

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 (tc_zhao@yahoo.com or xforms@world.std.com but not both). 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, the authors are aware of the following bindings

perl binding by Martin Bartlett (<martin@nitram.demon.co.uk>)

ada95 binding by G. Vincent Castellano (<gvc@ocsystems.com>)

Fortran binding by G. Groten (<zdv017@zam212.zam.kfa-juelich.de>) and Anke Haeming (<A.Haeming@kfa-juelich.de>)

pascal binding by Michael Van Canneyt (<michael@tfdec1.fys.kuleuven.ac.be>)

scm/guile binding by Johannes Leveling (<Johannes.Leveling@Informatik.Uni-Oldenburg.DE>)

python binding by Roberto Alsina (<ralsina@ultra7.unl.edu.ar>). (Seems the author has stopped working on this binding).

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://ncmir.ucsd.edu/pub/xforms
ftp://ftp.cs.ruu.nl/pub/XFORMS (Primary mirror site)
```

The primary site is mirrored by many sites around the world. The following are some of the mirror sites

```
ftp://ftp.fu-berlin.de/unix/X11/gui/xforms
ftp://gd.tuwien.ac.at/hci/xforms
ftp://ftp.st.ryukoku.ac.jp/pub/X11/xforms
ftp://ftp.via.ecp.fr/pub2/xforms
ftp://ftp.unipi.it/pub/mirror/xforms
ftp://ftp.uni-trier.de/pub/unix/X11/xforms
```

Additional mirrors, html version of this document, news and other information related to XForms can be accessed through www via the following URL

```
http://world.std.com/~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 unsubscribe, send a message to <xforms-request@bob.usuhs.mil> with one of the following commands as the mail body

```
help
subscribe
unsubscribe
```

To use the mailing list, send mail to <xforms@bob.usuhs.mil>. 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 <xforms-request@bob.usuhs.mil>.

The mailing list archive is at `http://bob.usuhs.mil/mailserv/list-archives`.

Thanks

Many people contributed, in one way or another, to the development of Forms Library, without whose testing, bug reports and suggestions, Forms Library would not be what it is today and would certainly not be in the relatively bug free state it is in now. We thank Steve Lamont of UCSD (<spl@szechuan.ucsd.edu>), for his numerous suggestions and voluminous contributions to the mailing list. We thank Erik Van Riper (<geek@midway.com>), formerly of CUNY, and Dr. Robert Williams of USUHS (<bob@bob.usuhs.mil>) for running the mailing list and keeping it running smoothly. We also thank every participant on the mailing list who contributed by asking questions and challenging our notion of what typical use of the Forms Library is. The html version of the document, undoubtedly browsed by the thousands, is courtesy of Danny Uy (<dau@westworld.com>). We appreciate the accurate and detailed bug reports, almost always accompanied with a demo program, from Gennady Sorokopud (<gena@NetVision.net.il>) and Rouben Rostamian (<rostamian@umbc.edu>). We also thank Martin Bartlett (<martin@nitram.demon.co.uk>), who, in addition to marrying Forms Library to perl, made several xforms API suggestions, Last but certainly not least, we thank Henrik Klagges (<henrik@UniX11.com>) for his numerous suggestions during the early stages of the development.

Part I - Using the Forms Library

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()], page 296`. 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()], page 296` 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 `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 [Goodies], page 69, 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 program that lets you interactively design forms and generate the corresponding C-code. See Chapter 7 [Introduction], page 82, and the following chapters 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 IV 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 [Part I Getting Started], page 9, we give some small and easy examples of the design and use of forms. In Chapter 3 [Defining Forms], page 15, we describe how to define forms. This chapter just contains the basic classes of objects that can be placed

[1] As of this writing, perl, Ada95, scheme, pascal, Fortran and python bindings are in beta testing.

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 [Doing Interaction], page 37, 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 [Free Objects], page 56, describes their use. Finally, Chapter 6 [Goodies], page 69, describes some built-in routines for simple interaction like asking questions and prompting for choices etc.

2 Getting Started

This chapter introduces the typographical conventions used throughout the manual and then continues with showing a few, simple examples on using the Forms Library. It concludes with a short resumee of the programming model typically found in programs using the library.

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 (at least) the first two letters are capitalized. For example,

```
FL_min()
FL_NORMAL_BUTTON
FL_OBJECT
```

The term "form" often can be taken to mean a window of your application. But be aware that there are also can be forms that themselves contain further forms, so "form" and "window" aren't necessarily synonyms.

The only exceptions from the above convention are names of functions related to image manipulations - they start with `flimage_`. And then there's a single function called `[flps_init()], page 291` that allows customization of the way hardcopies are created from an existing user interface.

2.2 Some Examples

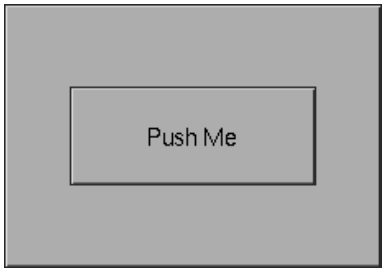
Before using forms for interaction with the user you first have to define them. Next you can display them and perform interaction with them. Both stages are simple. Before explaining all the details let us first look at some examples. A very simple form definition would look as

```
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. The form has a size of 230 by 160 pixels. 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 positioned in the form by virtue of the button geometry supplied and has "Push Me" as its label. After having defined the form we can display it using the call

```
fl_show_form(simpleform, FL_PLACE_MOUSE, FL_NOBORDER,
```

```
                "SimpleForm");
```



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 by calling

```
fl_do_forms();
```

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 `pushme.c` in the subdirectory `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

```
cc -o pushme pushme.c -lforms
```

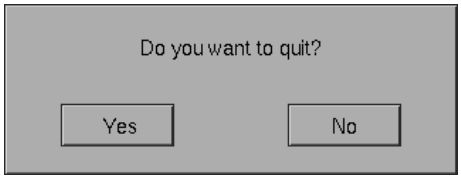
Please note that linking against the Forms library requires some other libraries to be istalled, at least the `X11` and the `Xpm` library. Some applications may also require the `JPEG` and/or the `GL` library. These libraries don't need to be specified explicitely in the linker command but must be available since the Forms library depends on them. If not installed contact your systems administrator.

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>

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

    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 (1) {
            if (fl_do_forms() == yes)
            {
                printf("Yes is pushed\n");
                break;
            }
            else
                printf("No is pushed\n");
        }

        fl_finish();
        return 0;
    }
```

It creates a form with a simple text and two buttons. After displaying the form `[fl_do_forms()], page 296` is called. This routine returns the object being pushed. Simply checking whether this is object `yes` or `no` determines whether we should quit.



As you see, the program starts by calling the routine `[fl_initialize()], page 277`. This routine should be called before any other calls to the library are made (except for `[fl_set_defaults()], page 279`). 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

Option	**Value type**	**Meaning**
`-display` *host:dpy*	string	Remote host
`-name` *appname*	string	change application name
`-visual` *class*	string	TrueColor, PseudoColor etc.
`-depth` *depth*	integer	Preferred visual depth
`-private`	none	Force a private colormap
`-shared`	none	Always share colormap
`-stdcmap`	none	Use standard colormap
`-fldebug` *level*	integer	Print some debug information
`-flhelp`	none	Print out these options

`-sync`	none	Force synchronous mode

Note that the executable name `argv[0]` should not contain period or `*`. See Chapter 32 [Overview of Main Functions], page 277, for further details. The above program can in fact be made a lot simpler, using the goodies described in Chapter 6 [Goodies], page 69. You can simply write:

```
while (!fl_show_question("Do you want to Quit?", 0))
    /* empty */ ;
```

Except printing out a message telling which button was pressed 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 is appropriate for simple programs. But, obviously, already for a program with just a few nmore objects it would become rather tedious to have to check each time `[fl_do_forms()], page 296` returns each of those objects to find out which of them was responsible and react accordingly. Utilizing object callback functions is then typically much easier and thus os strongly recommended.

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 code 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()], page 290`. After the form is realized (shown), control is handed to Forms Library's main loop `[fl_do_forms()], page 296`, 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 <stdio.h>
#include <stdlib.h>
#include <forms.h>

void yes_callback(FL_OBJECT *obj, long user_data) {
    printf("Yes is pushed\n");
    fl_finish();
    exit(0);
}

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

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

    return 0;
}
```

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()], page 290`. After the form is shown, the event handling is again handed to the main loop in Forms Library via `[fl_do_forms()], page 296`. 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()], page 296` never returns.

You might also have noticed that in this example both buttons are made anonymous, that is, it is not possible to reference the buttons outside of the creating 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.

The callback model presented above is the preferred way of interaction for typical programs and it is strongly recommended that programs using XForms be coded using object callbacks.

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, usually invoked by calling `[fl_do_forms()], page 296`, retrieves events from the X event queue, dispatches them to the appropriate objects and notifies the application of what action, if any, should be taken. The actual notification method depends on how the interaction is set up, which could be done by calling an object callback or by returning the object whose status has changed to the application program.

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

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 common 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 have to 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 some of 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). 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 of each form is a box. See the next section for the different types available. 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 some of the more common classes of objects that can be added to a form.

there's no built-in upper limit on the number of forms that can be defined and displayed when required. Normally you probably will first define all your forms before starting the actual work but it's no problem to define new forms also later on.

3.2 Boxes

The probably simplest type of objects are boxes. Boxes are used to give the forms and objects a nicer appearance. They can be used to visually group other objects together. The background 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);
```

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

`FL_NO_BOX`
: No box at all (it's transparent), just a label

`FL_UP_BOX`
: A box that comes out of the screen

`FL_DOWN_BOX`
: A box that goes down into the screen

`FL_BORDER_BOX`
: A flat box with a border

`FL_SHADOW_BOX`
: A flat box with a shadow

`FL_FRAME_BOX`
: A flat box with an engraved frame

`FL_ROUNDED_BOX`
: A rounded box

`FL_EMBOSSED_BOX`
: A flat box with an embossed frame

`FL_FLAT_BOX`
: A flat box without a border (normally invisible unless given a different color than the surroundings)

`FL_RFLAT_BOX`
: A rounded box without a border (normally invisible unless given a different color than the surroundings)

`FL_RSHADOW_BOX`
: A rounded box with a shadow

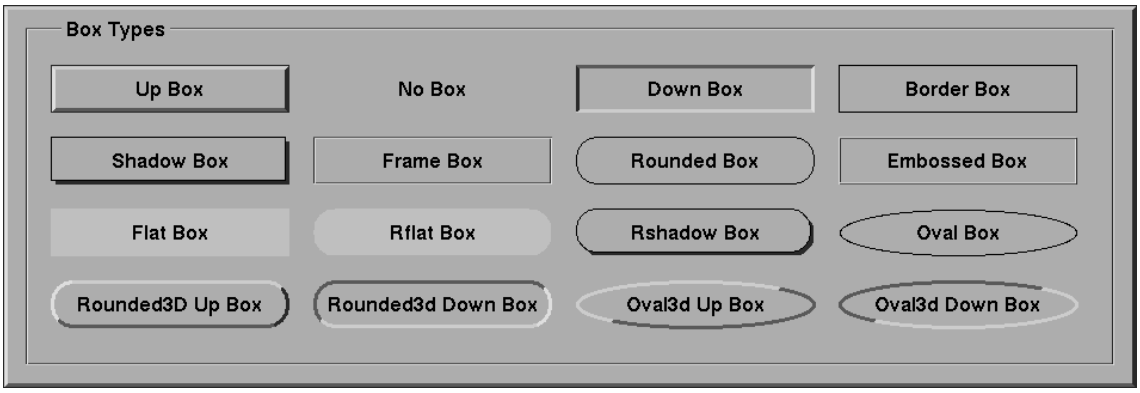
`FL_OVAL_BOX`
: A box shaped like an ellipse

`FL_ROUNDED3D_UPBOX`
: A rounded box coming out of the screen

`FL_ROUNDED3D_DOWNBOX`
: A rounded box going into the screen

`FL_OVAL3D_UPBOX`
: An oval box coming out of the screen

`FL_OVAL3D_DOWNBOX`
: An oval box going into the screen



The arguments x and y in the call of [fl_add_box()], page 109indicate the upper left corner of the box in the form while w and h are its width and height. 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 or a NULL pointer. 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 embedding <CNTRL> H (\010 or '\b') after the letter that needs to be underlined. If the very first character of the label is <Ctrl>H, the entire label is underlined.

The routine [fl_add_box()], page 109 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 [Changing Attributes], page 23. 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_lcolor(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 (but the functions for changing the object attributes can also used anywhere else within the program).

3.3 Texts

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);
```

where 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()], page 287 to be described in Section 3.11 [Changing Attributes], page 23. Again, the text can be multi-lined or underlined by embedding respectively the newline (\n) or <Ctrl>H (\010 or '\b') in the label. The style, size and color of the text can be controlled and changed in many ways, see Section 3.11.3 [Label Attributes and Fonts], page 27.

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 object. 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 3.11.3 [Label Attributes and Fonts], page 27. 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 get 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 [Doing Interaction], page 37, on the details of interaction).

A radio button is a push button with additional extra property: Whenever the user pushes a radio button, all other pushed radio buttons in the form (or at least in the group, see below) they belong to 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()], page 296` 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()], page 296` again after the appropriate action is performed; and in the latter, `[fl_do_forms()], page 296` would never return. See Chapter 4 [Doing Interaction], page 37, 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 appear pushed or not without user action. This is of course only useful for push buttons and radio buttons. To set or reset a push or radio button use the routine

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

`pushed` indicates whether the button should appear to be pushed (1) or released (0). Note that this does not invoke a callback routine bound to the button or results in the button getting returned to the program, i.e., only the visual appearance of the button is changed

and what it returns when asked for its state (and, in the case of a radio button, possibly that of another radio button in the same group). To also get the callback invoked or the button returned to the program additonally call e.g., [fl_trigger_object()], page 290.

To figure out whether a button appears as 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 [Changing Attributes], page 23.

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

Light buttons
: have a small "light" (colored area) 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()], page 120 with the same parameters as for 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()], page 287, see Section 3.11 [Changing Attributes], page 23.

Round buttons
: are buttons that are round. Use [fl_add_roundbutton()], page 120 to add a round button to a form.

Round3d buttons
: are buttons that are round and 3D-ish looking. Round and light buttons are nice as radio and push buttons.

Check buttons
: are buttons that have a small checkbox the user can push. To add a check button, use [fl_add_checkbutton()], page 120. More stylish for a group of radio buttons.

Bitmap buttons
: are buttons that have a bitmap on top of the box. Use routine [fl_add_bitmapbutton()], page 120 to add a bitmap button to a form.

Pixmap buttons
: are buttons that have a pixmap on top of the box. Use routine [fl_add_pixmapbutton()], page 120 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. 16.1 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);
```

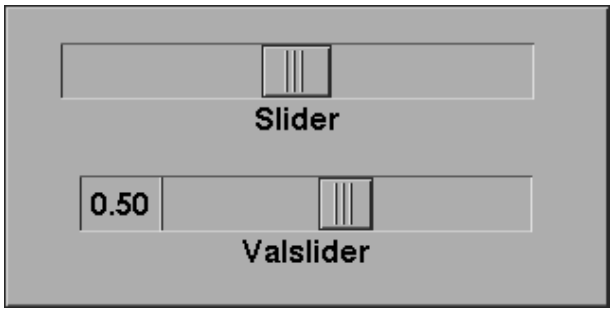
The two most important types of sliders are `FL_VERT_SLIDERM` and `FL_HOR_SLIDER`. The former displays a slider that can be moved vertically and the latter gives a slider that moves horizontally. In both cases the label is placed below the slider. Default value of the slider is 0.5 and can vary between 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

A 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.

To add a valslider, use

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

For all other interaction with a valslider the same function as for normal sliders can be used.

3.7 Input Fields

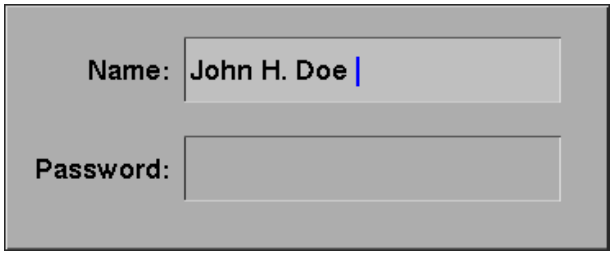
It is often required to obtain textual input from the user, e.g., a file name, some fields in a database, etc. To this end input fields exist in the Forms Library. An input field is a field that can be edited by the user using the keyboard. To add an input field to a form use

```
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()], page 287` to be described in Section 3.11 [Changing Attributes], page 23.

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 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`. To obtain the focus object, the following routine is available

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

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 routines

```
void fl_set_input(FL_OBJECT *obj, const char *str);
void fl_set_input_f(FL_OBJECT *obj, const char *fmt, ...);
```

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);
```

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

```
void fl_end_group(void);
```

The first function returns a pointer to a pseudo-object that represents the start of the group (its class is `FL_BEGIN_GROUP`). It can be used in a number of functions to work on the whole

group at once. Also the second creates a pseudo-object (of class `FL_END_GROUP`), marking the groups end, but since this object can't be used its address isn't returned.

Groups can't be nested. Groups are useful for two reasons. First of all it is possible to hide groups of objects. (see Section 3.9 [Hiding and Showing], page 22, below.) This is often very handy. We can, for example, display part of a form only when the user asks for it (see demo program `group.c`. 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

```
FL_OBJECT *fl_addto_group(FL_OBJECT *group);
```

where `group` is the object returned by `[fl_bgn_group()], page 285`. After this call, you can start adding objects to the group (e.g., `[fl_add_button()], page 120` etc.). The newly added objects are appended at the end of the group. When through with adding, use `[fl_end_group()], page 285` as before.

3.9 Hiding and Showing

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.

If you want to know if an object is shown you can use

```
int fl_object_is_visible(FL_OBJECT *obj);
```

Please note for an object to be visible also the form it belongs to must be shown, which isn't factored into the return value.

3.10 Deactivating 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);
```

To find out if an object is in active state use the function

```
int fl_object_is_active(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). This is not done automatically, so unless you set e.g., a different color the objects appearance won't change on deactivation (or re-activation).

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 *obj);
```

Calling this routine on an object `obj` results in the object returned to the application program or its callback being called if it exists. Note however, there is no visual feedback, i.e., `fl_trigger_object(button)` will not make the button object named `button` appear to be 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 depends on the type of the object. 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 has input focus, i.e., the user can enter text. 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.

To find out the colors of an object use

```
void fl_get_object_color(FL_OBJECT *obj,
                         FL_COLOR *col1, FL_COLOR *col2);
```

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

Name	**RGB triple**
FL_BLACK	(0, 0, 0)
FL_WHITE	(255, 255, 255),
FL_COL1	(173, 173, 173)
FL_BOTTOM_BCOL	(89, 89, 89)
FL_RIGHT_BCOL	(41, 41, 41)
FL_MCOL	(191, 191, 191)
FL_LEFT_BCOL	(222, 222, 222)
FL_LIGHTER_COL1	(204, 204, 204)
FL_DARKER_COL1	(161, 161, 161)
FL_SLATEBLUE	(113, 113, 198)
FL_INDIANRED	(198, 113, 113)
FL_RED	(255, 0, 0)
FL_BLUE	(0, 0, 255)
FL_GREEN	(0, 255, 0)
FL_YELLOW	(255, 255, 0)
FL_MAGENTA	(255, 0, 255)
FL_CYAN	(0, 255, 255)
FL_TOMATO	255, 99, 71
FL_INACTIVE	(110, 110, 110)
FL_TOP_BCOL	(204, 204, 204)
FL_PALEGREEN	(113, 198, 113)
FL_DARKGOLD	(205, 149, 10)
FL_ORCHID	(205, 105, 201)
FL_DARKCYAN	(40, 170, 175)
FL_DARKTOMATO	(139, 54, 38)
FL_WHEAT	(255, 231, 155)
FL_DARKORANGE	(255, 128, 0)
FL_DEEPPINK	(255, 0, 128)
FL_CHARTREUSE	(128, 255, 0)

FL_DARKVIOLET	(128, 0, 255)	
FL_SPRINGGREEN	(0, 255, 128)	
FL_DODGERBLUE	(0, 128, 255)	
FL_FREE_COL1	(?, ?, ?)	

Of all the colors listed in the table above FL_FREE_COL1 has the largest numerical value, and all color with 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()], page 277:

```
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_COLORS entries (default is 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()], page 284 before [fl_initialize()], page 277 to change XForms's default colormap. Note that 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()], page 277.

To change the colormap and make a color index active so that it can be used in various drawing routines after [fl_initialize()], page 277 initialization, use the following function

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

This function frees the previous allocated pixel corresponding to color index i and reallocates a pixel with the RGB value specified. The pixel value is returned by the function. It is recommended that you use an index larger than FL_FREE_COL1 for your remap request to avoid accidentally freeing the colors you have not explicitly allocated. Indices larger than 224 are reserved and should not be used.

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 name 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);
```

The function returns the pixel value as known by the Xserver. If the requested index, i, is never mapped or is freed, the RGB values as well as the pixel value are random. Since this function communicates with the Xserver to obtain the pixel information, it has

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

a two-way traffic overhead. If you're only interested in the internal colormap of XForms, [fl_get_icm_color()], page 284 is more efficient.

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 XForms 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()], page 256, resulting in accelerated colormap overflow and some other undesirable behavior. Since there may still be some 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);
```

Due to the use of an internal colormap and the simplified user interface, changing the colormap value for the 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

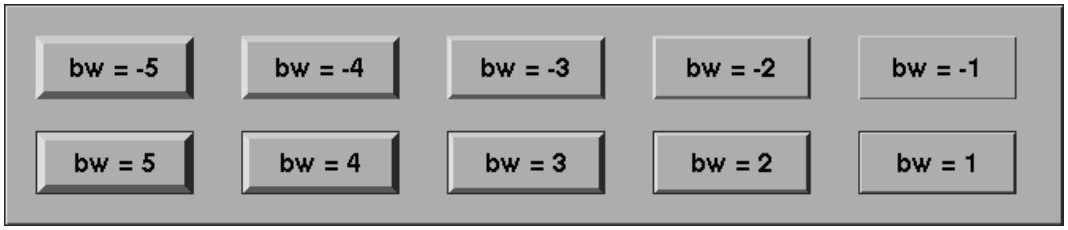
```
void fl_set_object_bw(FL_OBJECT *obj, int bw);
```

To find out about the current setting for the border width of an object call

```
int fl_get_object_bw(FL_OBJECT *obj);
```

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. The Forms Library's default is FL_BOUND_WIDTH (1) pixels (before version 1.0.91 the default was 3). 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. All applications developed using XForms accept a command line option '-bw', followed by an integer number,

the user can use to select the preferred border width. It is recommended that you document this flag in your application documentation. If you prefer a certain border width, use [fl_set_defaults()], page 279 or [fl_set_border_width()], page 281 before [fl_initialize()], page 277 to set the border width instead of hard-coding it on a per form or per object basis so the user has the option to change it at run time via the '-bw' flag.



There also exists a call that changes the object border width for the entire application

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

3.11.3 Label Attributes and Fonts

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

```
void fl_set_object_lcolor(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

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

where 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_SMALL_SIZE	10pt
FL_NORMAL_SIZE	12pt
FL_MEDIUM_SIZE	14pt
FL_LARGE_SIZE	18pt
FL_HUGE_SIZE	24pt

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`	
`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 48 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 32 fonts (numbers 16-47) can be filled in by the application program. Actually, the application program can also change the first 16 fonts if required (e.g., to force a particular resolution). To change a font for the the entire application, use one of the following routines:

```
int fl_set_font_name(int index, const char *name);
int fl_set_font_name(int index, const char *fmt, ...);
```

The first form accepts just a simple string for the font name while the second assembles the name from a format string as it's used with `printf()` etc. and the following arguments. The first argument, `index`, is the number of the font (between 0 and `FL_MAXFONTS-1`) and the font name should be a valid font name (with the exception of the size field). If you are defining a completely different font family starting at index `k`, it's a good idea to define `k + FL_BOLD_STYLE` to be the corresponding bold font in the family , and `k + FL_ITALIC_STYLE` the corresponding italic font in the family (so object like browser can obtain correct style when switching font styles):

```
#define Pretty       30
#define PrettyBold   (Pretty + FL_BOLD_STYLE)
#define PrettyItalic (Pretty + FL_ITALIC_STYLE)

fl_set_font_name(Pretty, fontname);
fl_set_font_name(PrettyBold, boldfontname);
fl_set_font_name(PrettyItalic, italicfontname);
...
fl_set_object_lstyle(obj, PrettyBold);
```

The function returns a negative value if the requested font is invalid or otherwise can't be loaded. Note however, if this routine is called before `[fl_initialize()], page 277`, it will return 0, but may fail later if the font name is not valid. To change the default font (helvetica-medium), a program should change font `FL_NORMAL_STYLE`.

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 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 while `shortform` 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()], page 283` or `[fl_set_font_name_f()], page 283` 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 should use the routine

```
void fl_set_object_lalign(FL_OBJECT *obj, int align);
```

with the following values for the `align` argument:

`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.

Alignment requests with the abpve constants place the text outside the box (except for `[FL_ALIGN_CENTER], page 30`). To get a value that can be used to align the label within the object the function

```
int fl_to_inside_lalign(int align);
```

can be used, which returns the necessary value for the corresponding inside alignment. Except for the case of `[FL_ALIGN_CENTER], page 30` (which is always inside the object) the result is the original value, logically or'ed with the constant .

There's also a function for the reverse conversion, i.e., from a calue for inside to outside alignment

```
int fl_to_outside_lalign(int align);
```

Using this functions is a bit simpler than combining the value with the `[FL_ALIGN_INSIDE], page 30` constant, especially when it comes to `[FL_ALIGN_CENTER], page 30` (which doesn't has the this bit set, even though labels with this alignment will always be shown within the object.

Both functions return `-1` if an invalid value for the alignment is passed to them.

There exist also three functions to test for the inside or outside alignment:

```
int fl_is_inside_lalign(int align);
int fl_is_outside_lalign(int align);
int fl_is_center_lalign(int align);
```

Note that these functions return `0` also in the case that the alignment value passed to them is invalid.

Not all objects accept all kinds of label alignment. For example for sliders, inputs etc. it doesn't make sense to have the label within the object und in these cases a request for an inside label is ignored (or, more precisely, converted to the corresponding request for an outside label or, on a request with `[FL_ALIGN_CENTER], page 30`, the reversion to the default label position). On the other hand, some objects like the text object (where the text to be shown is the label's text) accept only inside alignment and a request for an outside alignment will automatically replaced by the corresponding inside alignment.

See also the demo program `lalign.c` for an example of the positioning of labels using the above constants.

Finally, the routines

```
void fl_set_object_label(FL_OBJECT *obj, const char *label);
void fl_set_object_label_f(FL_OBJECT *obj, const char *fmt, ...);
```

change the label of a given object. Whilw the first function expects a simple string for the label. the second one accepts a format string with the same format specifiers as `printf()` etc., followed by as many additional arguments as there are format specifiers. An internal copy of the label for the object is made. As mentioned earlier, newline (`\n`) can be embedded in the label to generate multiple lines. By embedding `<Ctrl>H` (`\010`) in the label, the entire label or one of the characters in the label can be underlined. The function

```
const char * fl_get_object_label(FL_OBJECT *obj);
```

returns the label string.

3.11.4 Tool Tips

As will be seen later, an object can be decorated by icons instead of labels. For this kind of object, it is helpful to show a text string that explains the function the object controls under appropriate conditions. Forms Library elected to show the message after the mouse enters the object for about 600 milli-seconds. The text is removed when the mouse leaves the object or when the mouse is pressed.

To set the text, use the following routines

```
void fl_set_object_helper(FL_OBJECT *obj, const char *helpmsg);
void fl_set_object_helper_f(FL_OBJECT *obj, const char *fmt, ...);
```

where `helpmsg` is a text string (with possible embedded newlines in it) that will be shown when the mouse enters the object, after about a 600 milli-second delay. The second form of the function accepts instead a format string like `printf()` etc., followed by the appropriate number of arguments. In both cases an internal copy of the string is made.

The boxtype, color and font for the message display can be customized further using the following routines

```
void fl_set_tooltip_boxtype(int boxtype);
void fl_set_tooltip_color(FL_COLOR textcolor, FL_COLOR background);
void fl_set_tooltip_font(int style, int size);
void fl_set_tooltip_lalign(int align);
```

where `boxtype` is the backface of the form that displays the text. The default is `FL_BORDER_BOX`. `textcolor` and `background` specify the colors of the text string and the

backface. The defaults for these are `FL_BLACK` and `FL_YELLOW` respectively. The `style` and `size` parameters are the font style and size of the text. `align` is the alignment of the text string with respective to the box. The default is `FL_ALIGN_LEFT | FL_ALIGN_INSIDE` .

3.11.5 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 attributes change 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.

3.11.6 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, for example, just 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.

3.11.7 Symbols

Rather than using text as a label it is possible to place symbols like an arrows etc. on objects. This is done in the following way:

When the label starts with the character `@` instead of the text a particular symbol is drawn[2]. The rest of the label string indicates the symbol. A number of pre-defined symbols are available:

`->` Normal arrow pointing to the right.

[2] If you want a literal `@` character as the first character of a label text, escape it with another `@` character.

`<-`	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.
`>|`	A triangular arrow with a bar at the end.
`-->`	A thin arrow pointing to the right.
`=`	Three embossed lines.
`arrow`	Same as `-->`.
`returnarrow`	`<Return>` key symbol.
`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.
`UpArrow`	An embossed arrow.
`DnArrow`	An engraved arrow.

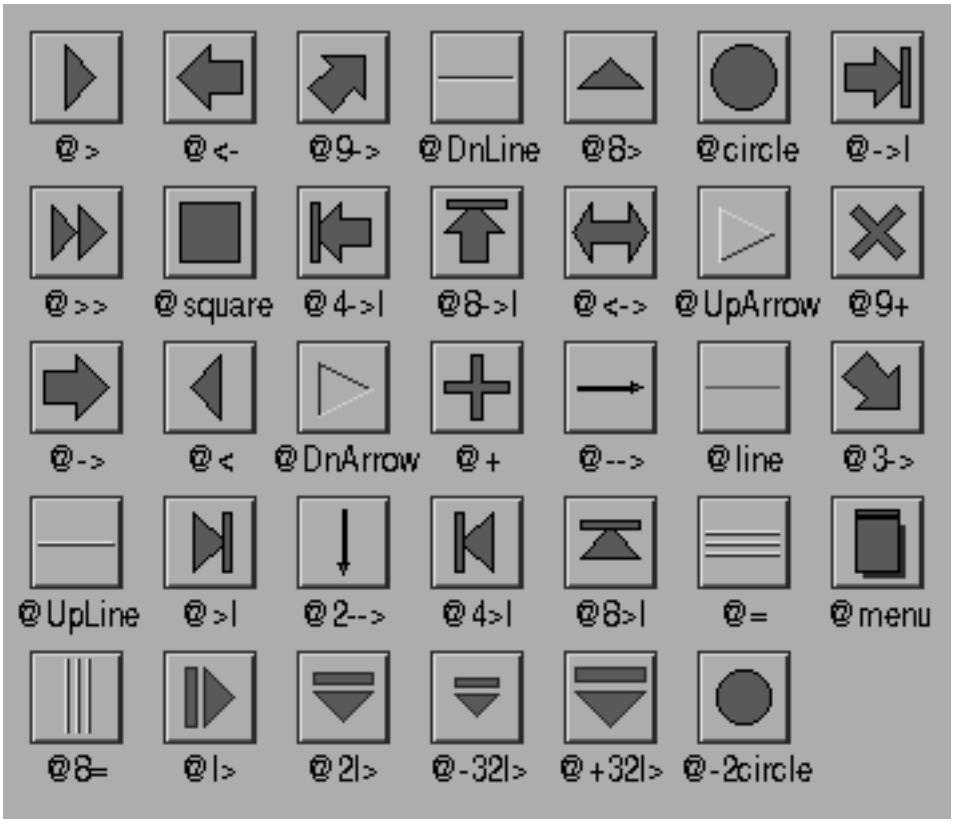
See Fig. 3.6 for how some of them look.

It is possible to use the symbols in different orientations. When the symbol name is preceded by a digit `1-9` it is rotated like on the numerical keypad, i.e., `6` (and also `5`) result in 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 with 6 pointing in the default orientation, i.e., to the right). 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 the angle (counter-clockwise). E.g., to draw an arrow at an angle of 30 degrees you can use `@030->`.

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. If keeping the aspect ratio is desired, put a sharp (`#`) immediately after the . E.g., `@#9->`.

Two additional prefixes, `+` and `-`, followed by a single digit, can be used to make small symbol size adjustment. A `+` indicates an increase of the symbol size while a `-` a decrease. The single digit following the prefix is the amount of increment (or decrement) in pixels. For example, to draw a square that is 3 pixels smaller in size than the default size use `@-3square`. If a single sequence of `+` or `-` and a single digit does not suffice, it can repeated, the effect is cumulative. Of course, this can also be combined with a rotation etc., so i.e., `@-9-3030->` (the order in which the different sequences are used doesn't matter) will result in an arrow drawn 12 pixels smaller than normal and rotated by 30 degrees counter-clockwise.

As already stated the "default" size of a symbol is (this at least holds for the built-in ones) one where it fits autoatically into the box it is to be drawn into, with a bit of room left around it. Thus the size of the symbol should in most cases be fine without any further fine-tuning. If you increase the size for whatever reasons please consider that the symbol automatically gets clipped to the area it is will be drawn into, i.e., increments that result in the symbol becoming larger than the box it is to be drawn into should be avoided.



In addition to using symbols as object labels, symbols can also be drawn directly using

```
int fl_draw_symbol(const char *symbolname, FL_Coord x, FL_Coord y,
                   FL_Coord w, FL_Coord h, FL_Color col);
```

(the function returns `1` on success and `0` on failure when the symbol name isn't valid) or indirectly via `[fl_drw_text()], page 263`. Drawing is clipped automatically to the area given by the arguments.

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, which may not have a `@`, a `#` or a digit at the start (or `+` or `-`, directly followed by a digit). `drawit()` is the routine to be called for drawing the symbol. `sc` is reserved and currently has no meaning. Best set it to `0`.

The routine `drawit()` should have the form

```
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 by `x`, `y`, `w`, `h` and rotated from its natural position by `angle` degrees. The draw function can call all types of drawing routines, including `[fl_draw_symbol()], page 34`. Before it is called clipping is set to the area given by the first four arguments.

If the new symbol name is the same as that of a built-in or of one previously defined, the new definition overrides the built-in or previously defined one.

The function returns `1` on success and `0` on failure (due to invalid arguments).

The symbol handling routines really should be viewed as a means of associating an arbitrary piece of text (the label) with arbitrary graphics, application of which can be quite pleasant given the right tasks.

A symbol (built-in or previously defined) can also be deleted using

```
int fl_delete_symbol(const char *name);
```

On success `1` is returned, otherwise `0`.

3.12 Adding and Removing Objects

In some situations you might want to add objects to an already existing form (i.e., a form for which `fl_end_form()` has already been called. Reopening a form for the addition of further objects can be done by using the call

```
FL_FORM *fl_addto_form(FL_FORM *form);
```

After this call you can again add objects to the form with the usual functions for adding objects (like `[fl_add_button()], page 120` etc.). When done with adding objects to the form again call `[fl_end_form()], page 285`. 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 remove an object from a form simply use

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

It removes the object from the form it currently belongs to and also from a group it may belong to. The argument can also be the pseudo-object starting a group (i.e., the return value of `[fl_bgn_group()], page 285`) in which case the whole group of objects will be removed from the form.

Contrary to what the name of the function may hint at the object itself isn't deleted but it remains available (except if it's an object that marks the start or end of a group) and thus it can be added again to the same or another form (without having to call `[fl_addto_form()], page 286` first and `[fl_end_form()], page 285` afterwards) using the function

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

Normally, this function should only be used within object classes to add a newly created object to the form currently under construction. It can not be used for pseude-objects representing the start or end of a group.

3.13 Freeing Objects

If the application program does not need an object anymore it can completely delete it, freeing all memory used for it, using a call of

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

After this the object is truely destroyed and can no longer be used. If you hadn't removed the object from the form it did belong to using `[fl_delete_object()], page 286` before this will be done automatically.

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

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

This will delete and free all the objects of the form and the form itself. A freed form can not be referenced anymore.

4 Doing Interaction

4.1 Displaying a Form

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 parameter `name` is the title of the form (and its associated icon if any). The routine returns the ID of the forms window. 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()], page 292` 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 the following function after `[fl_show_form()], page 292`

```
void fl_update_display(int blocking);
```

where `blocking` is false (0), the function flushes the X buffer so the drawing requests are on their way to the server. When `blocking` is true (1), the function flushes the buffer and waits until all the events are received and processed by the server. For typical programs that use `[fl_do_forms()], page 296` or `[fl_check_forms()], page 296` after `[fl_show_form()], page 292`, flushing is not necessary as the output buffer is flushed automatically. Excessive call to `[fl_update_display()], page 37` degrades performace.

The location and size of the window to be shown on the call of `[fl_show_form()], page 292` are determined by the `place` argument. 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()], page 294`. 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()], page 294`. 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], page 38` as in `[FL_PLACE_MOUSE], page 37|[FL_FREE_SIZE], page 38`.

`FL_PLACE_CENTER`
: The form is placed in the center of the screen. If [FL_FREE_SIZE], page 38 is also specified, interactive resizing will be allowed.

`FL_PLACE_FULLSCREEN`
: The form is scaled to cover the full screen. If [FL_FREE_SIZE], page 38 is also specified, interactive 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 setvia [fl_set_form_position()], page 294, will be used otherwise interactive positioning may be possible if the window manager allows it.

`FL_PLACE_HOTSPOT`
: The form is placed so that mouse is on the form's "hotspot". If [FL_FREE_SIZE], page 38 is also specified, interactive resizing will be allowed.

`FL_PLACE_CENTERFREE`
: Same as [FL_PLACE_CENTER], page 37|[FL_FREE_SIZE], page 38, 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.

As mentioned above, some of the settings will result in a fixed size of the form (i.e., a size that can't be changed by the user per default). In some cases this can be avoided by OR'ing the value with `FL_FREE_SIZE` as a modifier.

If no size was 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 allow interactive placement of the windows but some don't.

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, combining both these two functions,

```
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. They will change the position and/or size of the form.) `x`, `y`, `w` and `h` indicate the position of the form on the screen and its size[1]. 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

[1] The parameters should be sensitive to the coordinate unit in effect at the time of the call, but at present, they are not, i.e., the function takes only values in pixel units.

[FL_PLACE_GEOMETRY], page 37, [FL_PLACE_FREE], page 38. E.g., to place a form at the lower-right corner of the screen use

```
fl_set_form_position(form, -1, -1);
fl_show_form(form, FL_PLACE_GEOMETRY, FL_TRANSIENT, "formName");
```

(Following the X convention for specifying geometries a negative x-position specifies the distance of the right eside of the form from the right side of the screen and a negative y-position the distance of the bottom of the form from the bottom of the screen.)

To show a form so that a particular object or point is under 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 *obj);
```

and then use [FL_PLACE_HOTSPOT], page 38 for the place argument in the call of [fl_show_form()], page 292. The coordinates x and y are relative to the upper-left hand corner of the form (within the window decorations).

In the call [fl_show_form()], page 292 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 decorations.

FL_TRANSIENT
: Borders with (possibly) less decorations.

FL_NOBORDER
: No decoration at all.

For some dialogs, such as demanding an answer etc., you probably do not want the window manager's full decorations. Use [FL_TRANSIENT], page 39 for this.

A window border is useful to let the user iconify a form, move it around or resize it. 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 the window managers as well as their options. [FL_NOBORDER], page 39 is guaranteed to have no border[2] and is immune to iconification request. Because of this, borderless forms can be hostile to other applications[3], 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()) 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

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

[2] Provided the window manager is compliant. If the window manager isn't compliant all bets are off.

[3] Actually, they are also hostile to their sibling forms. See Chapter 32 [Overview of Main Functions], page 277.

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()], page 292` 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()], page 292`.

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 per default will also be scaled. This includes both the sizes and positions of the objects. For most cases, this default behavior is adequate. In some cases, e.g., to keep a group of objects together, more control is needed. To this end, the following routines can be used

```
void fl_set_object_resize(FL_OBJECT *obj, unsigned how_resize);
void fl_set_object_gravity(FL_OBJECT *obj,
                           unsigned nw_gravity, unsigned se_gravity);
```

The `how_resize` argument of `[fl_set_object_resize()], page 289` can be one of

`FL_RESIZE_NONE`
: don't resize the object at all

`FL_RESIZE_X`
: resize it in x- (horizontal) direction only

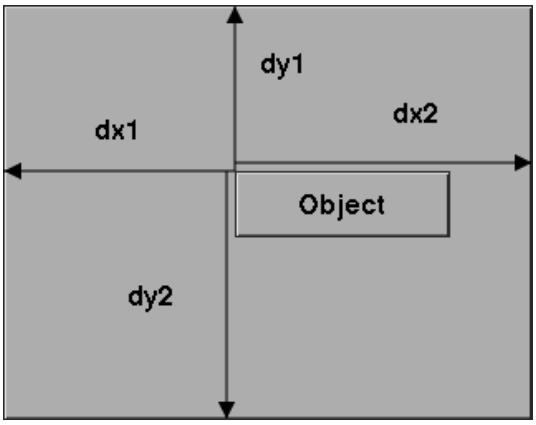
`FL_RESIZE_Y`
: resize it in y- (vertical) direction only

`FL_RESIZE_ALL`
: is an alias for `[FL_RESIZE_X], page 40|[FL_RESIZE_Y], page 40` and makes the object resizable in both dimension.

The arguments `nw_gravity` and `se_gravity` of `fl_set_object_gravity()` control 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
`ForgetGravity`	don't consider the setting for this argument



Default for all object is `[FL_RESIZE_ALL], page 40` and `[ForgetGravity], page 41`. Note that the three parameters are not orthogonal and the positioning request will always override the scaling request in case of conflict. This means the resizing settings for an object are considered only if one (or both) of the gravities is `[FL_NoGravity], page 40`.

For the special case where `how_resize` is `[FL_RESIZE_NONE], page 40` and both gravities are set to `ForgetGravity`, 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], page 37` or other size-restricting options, resizing it later via `[fl_set_form_size()], page 294` will likely be rejected.

To determine the gravity and resize settings for an object use the functions

```
void fl_get_object_gravity(FL_OBJECT *obj,
                           unsigned int *nw, unsigned int *se);
void fl_get_object_resize(FL_OBJECT *obj, unsigned int *resize );
```

Sometimes, you may want to change an attribute for all objects on a particular form, to this end, the following iterator is available

```
void fl_for_all_objects(FL_FORM *form,
                        int (*operate)(FL_OBJECT *obj, void *data),
                        void *data);
```

where function `operate` is called for every object of the form `form` unless `operate()` returns nonzero, which terminates the iterator.

Multiple forms can be shown at the same moment and the system will interact with 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 Part V for details.

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

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

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 Section 15.6.4 [Other Pixmap Routines], page 114, for some of the routines that can be used to create Pixmaps.) Note that an `icon` previously setvia this function (if it exists) is not freed or modified in anyway. See the demo program `iconify.c` for an example.

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

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

To check if a form is visible or not, use the following call

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

The function returns one of

`FL_INVISIBLE`
: if the form is not visible (0),

`FL_VISIBLE`
: if the form is visible (1) and

`FL_BEING_HIDDEN`
: if the form is visible but is in the process of being hidden (-1).

Note that if you don't need a form anymore you can deallocate its memory using the call `[fl_free_form()], page 286` 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 message, 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 record keeping or to elect to ignore this message, register a callback function using the following routine

```
int 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 the constant `FL_IGNORE`. Any other value (e.g., `FL_OK`) will result in the termination of the application.

Similar mechanism exists for individual forms

```
int 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. Of course, if you'd like to terminate the application, you can always call `exit(3)` yourself within the callback function.

4.2 Simple Interaction

Once one or more forms are shown it is time to give 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 most programs, is to call of

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

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()], page 296`.

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 per default is returned whenever its value is changed, so whenever the user clicks on it and moves the mouse the slider object gets returned.

input — An input field is returned per default when it is deactivated, i.e., the user has selected it and then starts interacting with another object that has the ability to get returned.

(This list just contains a small number of objects that exist, see Part III for a list of all objects and the documentation of the exact behaviour of them.)

When the (address of the) object is returned by `[fl_do_forms()], page 296` the application program can 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()],`

page 296 is called again to continue the interaction. Hence, simpler programs have the following global form:

```
/* define the forms */
/* display the forms */
while (! ready) {
    obj = fl_do_forms();
    if (obj == obj1)
        /* handle the change in obj1 */
    else if (obj == obj2)
        /* handle the change in obj2 */
    ....
}
```

For more complex programs interaction via callbacks is often preferable. For such programs, the global structure looks something like the following

```
/* define callbacks */
void callback(FL_OBJECT *obj, long data) {
    /* perform tasks */
}

void terminate_callback(FL_OBJECT *obj, long data) {
    /* cleanup application */
    fl_finish();
    exit(0);
}

main(int argc, char *argv[]) {
    /* create form and bind the callbacks to objects */
    /* enter main loop */
    fl_do_forms();
    return 0;
}
```

In this case, [fl_do_forms()], page 296 handles the interaction indefinitely and never returns. The program exits via one of the callback functions.

There is also the possibility to conrol under which exact conditions the object gets returned. An application that e.g., doesn't want to be notified about each change of a slider but instead only want a single notification after the mouse button has been released and the value of the slider was changed in the process would call the function

```
int fl_set_object_return(FL_OBJECT *obj, unsigned int when);
```

with when set to FL_RETURN_END_CHANGED.

There are several values when can take:

FL_RETURN_CHANGED
: Return (or call object callback) whenever there is a change in the state of the object (button was pressed, input field was changed, slider was moved etc.).

`FL_RETURN_END`
: Return (or invoke callback) at the end of the interaction (typically when the user releases the mouse button) regardless if the objects state was changed or not.

`FL_RETURN_END_CHANGED`
: Return (or call object callback) when interaction stops **and** the state of the object changed.

`FL_RETURN_SELECTION`
: Return when e.g., a line in a [FL_MULTI_BROWSER], page 169 browser was selected.

`FL_RETURN_DESELECTION`
: Return when e.g., a line in a [FL_MULTI_BROWSER], page 169 browser was deselected.

`FL_RETURN_ALWAYS`
: Return (or invoke callback) on any of the events that can happen to the object.

`FL_RETURN_NONE`
: Never notiy the application about interactions with this object (i.e., never return it nor invoke its callback). Note: this is not meant for deactivation of an object, it will still seem to work as normal, it just doesn't get returned to the application nor does is callbak get invoked.

Since for different objects only subsets of these conditions make sense please read the more detailed descriptions for each of the object types in Part III.

All of the values above, except [FL_RETURN_END_CHANGED], page 45, [FL_RETURN_ALWAYS], page 45 and [FL_RETURN_NONE], page 45 can be logically `OR`'ed. [FL_RETURN_END_CHANGED], page 45 is different in that it only can be returned when the conditions for [FL_RETURN_END], page 44 and [FL_RETURN_CHANGED], page 44 are satisfied at once. If this is request [FL_RETURN_END], page 44 and [FL_RETURN_CHANGED], page 44 will automatically deselected. So if you want notifications about the conditions that lead to [FL_RETURN_END], page 44 or [FL_RETURN_CHANGED], page 44 (or both at once) ask instead for the logical `OR` of these two.

[FL_RETURN_ALWAYS], page 45 includes all conditions except [FL_RETURN_END_CHANGED], page 45.

Once an object has been returned (or its callback is invoked) you can determine the reason why it was returned by calling

```
int fl_get_object_return_state(FL_OBBJECT *obj);
```

This returns the logical `OR` of the conditions that led to the object being returned, where the conditions can be [FL_RETURN_CHANGED], page 44, [FL_RETURN_END], page 44, [FL_RETURN_SELECTION], page 45 and [FL_RETURN_DESELECTION], page 45. (The [FL_RETURN_END_CHANGED], page 45 condition is satisfied if both [FL_RETURN_END], page 44 and [FL_RETURN_CHANGED], page 44 are set.)

Please note that calling this function only makes sense in a callback for an object or when the object has been just returned by e.g., [fl_do_forms()], page 296. Further interactions with the object overwrite the value!

4.3 Periodic Events and Non-blocking Interaction

The 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), some other means of interaction are needed.

There exist two different, but somewhat similar, mechanisms in the library that are designed specifically for generating and handling periodic events or achieving non-blocking interaction. Depending on the application, one method may be more appropriate than the other.

For periodic tasks, e.g., rotating an image, checking the status of some external device or application state etc., interaction via an idle callback comes in very handy. An idle callback is an application function that is registered with the system and is called whenever there are no events pending for forms (or application windows).

To register an idle callback, use the following routine

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

After the registration, whenever the main loop (`[fl_do_forms()], page 296`) is idle, i.e., no user action or light user action, the callback function of type `FL_APPEVENT_CB` is called

```
typedef int (*FL_APPEVENT_CB)(XEvent *xev, void *user_data);
```

i.e., a function with the signature

```
int idle_callback(XEvent *xev, void *user_data);
```

where `user_data` is the void pointer passed to the system in `[fl_set_idle_callback()], page 300` through which some information about the application can be passed. The return value of the callback function is currently not used. `xev` is a pointer to a synthetic[4] `MotionNotify` event from which some information about mouse position etc. can be obtained. To remove the idle callback, use `[fl_set_idle_callback()], page 300` with callback set to `NULL`.

Timeouts are similar to idle callbacks but with somewhat more accurate timing. Idle callbacks are called whenever the system is idle, the time interval between any two invocations of the idle callback can vary a great deal depending upon many factors. Timeout callbacks, on the other hand, will never be called before the specified time is elapsed. You can think of timeouts as regularized idle callbacks, and further you can have more than one timeout callbacks.

To add a timeout callback, use the following routine

```
typedef void (*FL_TIMEOUT_CALLBACK)(int, void *);
int fl_add_timeout(long msec, FL_TIMEOUT_CALLBACK callback,
                   void *data);
```

The function returns the timeout's ID[5]. When the time interval specified by `msec` (in milliseconds) has elapsed the timeout is removed, then the callback function is called. The timeout ID is passed to the callback function as the first parameter. The second parameter

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

[5] The function will not return 0 or -1 as timeout IDs, so the application program can use these values to tag invalid or expired timeouts.

the callback function is passed is the data pointer that was passed to [fl_add_timeout()], page 300.

To remove a timeout before it triggers, use the following routine

```
void fl_remove_timeout(int id);
```

where id is the timeout ID returned by [fl_add_timeout()], page 300. There is also an FL_OBJECT, the FL_TIMER object, especially the invisible type, that can be used to do timeout. Since it is a proper Forms Library object, it may be easier to use simply because it has the same API as any other GUI elements and is supported by the Form Designer. See Section 21.1 [Timer Object], page 186, for complete information on the FL_TIMER object.

Note that idle callback and timeout are not appropriate for tasks that block or take a long time to finish because during the busy or blocked period, no interaction with the GUI can take place (both idle callback and timeout are invoked by the main loop, blockage or busy executing application code prevents the main loop from performing its tasks).

So what to do in situations where the application program does require a lengthy computation while still wanting to have the ability to interact with the user interface (for example, a Stop button to terminate the lengthy computation)?

In these situations, the following routine can be used:

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

This function is similar to [fl_do_forms()], page 296 in that it takes care of handling events and appropriate callbacks, but it does not block. Instead it always returns to the application program immediately. If a change has occurred in some object the object is returned as with [fl_do_forms()], page 296. But when no change has occurred control is also returned but this time a NULL object is returned. Thus, by inserting this statement in the middle of the computation in appropriate places in effect "polls" the user interface. The downside of using this function is that if used excessively, as with all excessive polls, it can chew up considerable CPU cycles. Therefore, it should only be used outside the inner most loops of the computation. If all objects have callbacks bound to them, [fl_check_forms()], page 296 always returns NULL, otherwise, code similar to the following is needed:

```
obj = fl_check_forms();
if (obj == obj1)
    /* handle it */
...
```

Depending on the applications, it may be possible to partition the computation into smaller tasks that can be performed within an idle callback one after another, thus eliminating the need of using [fl_check_forms()], page 296.

Handling intensive computation while maintaining user interface responsiveness can be tricky and by no means the above methods are the only options. You can, for example, fork a child process to do some of the tasks and communicate with the interface via pipes and/or signals, both of which can be handled with library routines documented later, or use multi-thread (but be careful to limit Xserver access within one thread). Be creative and have fun.

For running external executables while maintaining responsiveness of the interface, see [fl_exe_command()], page 73 and [fl_popen()], page 73 documented later in Section 6.2 [Command Log], page 73.

4.4 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()], page 296 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_lcolor()], page 288 (see objinactive.c. What objects to gray out is obviously application dependent.

The following two functions can be used to register two callbacks 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 functions:

```
void fl_deactivate_all_forms(void);
void fl_activate_all_forms(void);
```

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()], page 296 and [fl_check_forms()], page 296 read this queue and handle the events. When the application program itself also opens windows, these windows will rather likely receive events as well. Unfortunately, there is only one event queue. When both the application program and the library routines would read events from this one queue problems would occur and events missed. Hence, the application program should not read the event queue itself.. To solve this problem, the library 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 must use replacements for the usual Xlib routines. Instead of using `XNextEvent()`, the program will use `[fl_XNextEvent()], page 49`, with the same parameters except the `Display *` argument. The following is a list of all replacement routines:

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

Note that these routines normally return `1`, but after a call of `[fl_finish()], page 285` they return `1` instead.

Other events routines may be directly used if proper care is taken to make sure that only events for the application windows not handled by the library are removed. These routines include `XWindowEvent()`, `XCheckWindowEvent()` etc.

To help find out when an event has occurred, whenever `[fl_do_forms()], page 296` and `[fl_check_forms()], page 296` encounter an event that is not meant for handling by the library but by the application program itself they return a special object `[FL_EVENT], page 300`. Upon receiving this special event, the application program can and must remove the pending event from the queue using `[fl_XNextEvent()], page 49`.

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 may not want to receive these "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 not want to discard any events. In this case you can use the routines `[fl_do_only_forms()], page 296` and `[fl_check_only_forms()], page 296` that will never return `[FL_EVENT], page 300`. The events don't disappear but will be returned at later calls to the normal routines `[fl_do_forms()], page 296` etc.

It can't be over-emphasized that it is an error to ignore `[FL_EVENT], page 300` or use `[fl_XNextEvent()], page 49` without seeing `[FL_EVENT], page 300`.

Sometimes an application program might need to find out more information about the event that triggered a callback, e.g., to implement mouse button number sensitive functionalities. To this end, the following routines may be called

```
long fl_mouse_button(void);
```

This function, if needed, should be called from within a callback. The function returns one of the constants [FL_LEFT_MOUSE], page 243, [FL_MIDDLE_MOUSE], page 243, [FL_RIGHT_MOUSE], page 243, [FL_SCROLLUP_MOUSE], page 243 or [FL_SCROLLDOWN_MOUSE], page 243, indicating which mouse button was pushed or released. If the callback is triggered by a shortcut, the function returns the keysym (ascii value if ASCII) of the key plus [FL_SHORTCUT], page 244. For example, if a button has a shortcut <Ctrl>C (ASCII value is 3), the button number returned upon activation of the shortcut would be FL_SHORTCUT + 3. [FL_SHORTCUT], page 244 can be used to determine if the callback is triggered by a shortcut or not

```
if (fl_mouse_button() >= FL_SHORTCUT)
    /* handle shortcut */
else
    switch (fl_mouse_button()) {
        case FL_LEFTMOUSE:
        ....
    }
```

More information can be obtained by using the following routine that returns the last XEvent

```
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);
```

4.5 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()], page 296` or `[fl_check_forms()], page 296` 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 *obj, 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. The 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()], page 290`.

See program `yesno_cb.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 three fields, `void *u_vdata`, `char *u_cdata` 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. A better and more general solution to the problem is detailed in Part II of this documentation where all objects on a form is are grouped into a single structure which can then be "hang" off of `u_vdata` or some field in the `FL_FORM` structure.

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()], page 124`, `[fl_set_input()], page 152` 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()], page 296 or [fl_check_forms()], page 296 would return an object in form they call the routine callback instead, with the object as an argument. So the callback should have the form

```
void callback(FL_OBJECT *obj, void *data);
```

With each form you can associate its own callback routine. For objects that have their own callbacks the object callbacks have priority over the form callback.

When the application program also has its own windows (via Xlib or Xt), it most likely also wants to know about XEvents for the window. As explained earlier, this can be accomplished by checking for [FL_EVENT], page 300 objects. Another (and better) 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 (of type [FL_APPEVENT_CB], page 46 use the call

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

Whenever an event happens the callback function is invoked with the event as the first argument and a pointer to data you want it to receive. So the callback should have the form

```
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. The callback function normally should return 0 unless the event isn't for one of the applcation-managed 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 that the callback routine will handle all events for the window. The newly installed callback replaces the callback already installed. Note that this function only works for windows created directly by the application program (i.e., it won't work for forms' windows or windows created by the canvas object). It is possible to access the raw events that

happen on a form's window via [fl_register_raw_callback()], page 314 discussed in Section 35.1.1 [Form Events], page 314.

[fl_add_event_callback()], page 297 does not alter the window's event mask nor does it solicit events for you. That's 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);
```

This function activates the default mapping of events to event masks built-in in the Forms Library, and causes the system to solicit the events for you. Note however, the mapping of events to masks are not unique and depending on applications, the default mapping may or may not be the one you want. For example, MotionNotify event can be mapped into ButtonMotionMask or PointerMotionMask. Forms Library will use both.

It is possible to control the masks you want precisely by using the following function, which can also be used 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. If event callback functions are registered via both fl_set_event_callback() and [fl_add_event_callback()], page 297, the callback via the latter is invoked first and the callback registered via [fl_set_event_callback()], page 297 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);
```

and all Expose events on window winID are consumed by expose_cb then 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()], page 297. If xev_type is 0, all callbacks for window win are removed. This routine is called automatically if [fl_winclose()], page 305 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) {
    /* 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 */
    fl_addto_selected_xevent(winID,
                             ExposureMask | ButtonPressMask |... );
    fl_winshow(winID);
    fl_set_event_callback(event_cb, whatever);
    fl_add_event_callback(winID, Expose, expose_cb, data);
    fl_do_forms();
    return 0;
}
```

The routine `[fl_do_forms()], page 296` 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.

4.6 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 occurred.

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.

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 or use a canvas (the preferred way) in which it does interaction with the user). A second way is to add your own object classes (see Part IV). 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 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_KEYPRESS` 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()], page 296` or `[fl_check_forms()], page 296` 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()], page 254 can be used as an index into a table of structures, [fl_state], page 301[], from which all information about current active visual can be obtained. See Chapter 28 [Drawing Objects], page 254, 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 FL_PUSH event was sent earlier.

FL_DBLCLICK
: The user has pushed a mouse button twice within a certain time limit (FL_CLICK_TIMEOUT), which by default is about 400 msec.

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_UPDATE
: The mouse position has changed. This event is sent to an object between an FL_PUSH and an FL_RELEASE event (actually this event is sent periodically, even if mouse has not moved). The mouse position is given as the parameter mx and my and action can be taken based on the position.

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

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_KEYPRESS
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()], page 297.

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_KEYPRESS events. The obj->wantkey is by default set to FL_KEY_NORMAL, i.e., the free object will receive all normal keys (0-255) except <Tab> and <Return> key. If you're interested in <Tab> or <Return> key, you need to change obj->wantkey to FL_KEY_TAB or FL_KEY_ALL. See Chapter 26 [Events], page 242, for details.

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 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 * miscgrp;
    FL_OBJECT * sizegrp;
    FL_OBJECT * wsli;
    FL_OBJECT * hsli;
    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 objects are 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_calloc(1, sizeof *fdui);

    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_lcolor(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_lcolor(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_lcolor(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 the callback function `switch_figure()` is bound to all three buttons but with different arguments. Thus the callback function can resolve the associated object via 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 the amount of code 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 these sliding bars (red, green, blue), there is no need to label them. There's also no need to store their addresses as their callback routine `change_color()` will receive them automatically.

```
    fdui->colgrp = fl_bgn_group();

    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);

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);

 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 the color indexed by `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 buttons. 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 purposes 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 of 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, two sliders are added, using the same callback function but 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 outlined instead of filled figures etc. The form definition ends with `[fl_end_form()], page 285`. The structure that holds the form as well as all the objects within it 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,        /* is it to be filled? */
             x, y, w, h;  /* position and sizes */
    int      pc[3];       /* primary color R,G,B */
    int      newfig;      /* indicate a new figure */
    FL_COLOR col;         /* color index */
} DrawFigure;

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

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 routines are 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()], page 259 returns the current "active" window, defined to be the window the object receiving the dispatcher's messages (FL_DRAW etc.) belongs to[2]. Similarly the routine [fl_get_display()], page 255 returns the current connection to the X server. Part IV has more details on the utility functions in the Forms Library.

The array of structures [fl_state], page 301[] 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 *obj, long which) {
    cur_fig->c[which] = 255 * fl_get_slider_value(obj);
    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 call of [fl_mapcolor()], page 256 defines the RGB components for index cur_fig->col and the second [fl_mapcolor()], page 256 call 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 * obj, long which) {
    if (which == 0)
        cur_fig->w = fl_get_slider_value(obj);
    else
        cur_fig->h = fl_get_slider_value(obj);
```

[2] If [fl_winget()], page 259 is called while not handling messages, the return value must be checked.

```
}
```

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

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

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 *obj, 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 + obj->x,
                                 cur_fig->y + obj->y,
                                 cur_fig->w, cur_fig->h,
                                 cur_fig->col);
            else {
                fl_drw_box(obj->boxtype, obj->x, obj->y, obj->w,
                           obj->h, obj->col1, obj->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 + obj->x,
                               dr->y + obj->y,
                               dr->w, dr->h, dr->col);
                }
            }
```

```
                cur_fig->newfig = 0;
                break;

            case FL_PUSH:
                if (key == FL_MIDDLE_MOUSE)
                    break;

                cur_fig->x = mx - cur_fig->w / 2;
                cur_fig->y = my - cur_fig->h / 2;

                /* convert figure center to relative to the object*/
                cur_fig->x -= obj->x;
                cur_fig->y -= obj->y;

                cur_fig->newfig = 1;
                fl_redraw_object(obj);
                *(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 FL_RETURN_NONE;
    }
```

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 anything else than `FL_DRAW`. So always use `[fl_redraw_object()], page 297` to draw unless it is a response to `FL_DRAW`. Internally `[fl_redraw_object()], page 297` 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()], page 296` will return only if that button is pushed. Full source code to this simple drawing program can be found in `demos/freedraw.c`.

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 three lines of text and a button labeled OK on it. The form is so shown such that the mouse pointer is on the button.

Sometimes, it may be more convenient to use the following routine

```
void fl_show_messages(const char *str);
```

when the message is a single line or when you know the message in advance. Embed newlines in `str` to get multi-line messages.

As a third alternative you can also use

```
void fl_show_messages_f(const char * fmt, ...);
```

The only required argument `fmt` is a format string as you would use it for e.g., `printf(3)`, which then is followed by as many arguments as there are format specifiers in the format string. The string resulting from expanding the format string, using the remaining arguments, can have arbitrary length and embedded newline characters (`'\n'`), producing line breaks. The size of the message box is automatically made to fit the whole text.

Both of the message routines block execution and do not return immediately (but idle callbacks and asynchronous IO continue to be run and checked). Execution resumes when the OK button is pressed or `<Return>` is hit, or when the message form is removed from the screen by the following routine (for example, triggered by a timeout or idle callback):

```
void fl_hide_message(void)
```

There is also a routine that can be used to show a 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 coordinates (relative to the root window) the message should be placed. Note that multi-line messages are possible by embedding the newline character into `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);
```

A similar routine exists to change the font style and size

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

```
void fl_show_alert(const char *s1, const char *s2, const char *s3,
                   int centered);
void fl_hide_alert(void);
```

work the same as [fl_show_messages()], page 69 goodie except that an alert icon (!) is added and the first string is shown bold-faced. The extra parameter `centered` controls whether to display the form centered on the screen.

As in the case of messages also another function is avaialble

```
void fl_show_alert2(int centered, const char *fmt, ...);
```

`centered` controls if the alert message is centered and `fmt` must be a format string as e.g., used for `printf(3)`. After the format string as many further arguments are required as there are format specifiers in the format string. The string resulting from expanding the format string, using the rest of the arguments, can have arbitrary length and the first embedded form-feed character (`'\f'`) is used as the separator between the title string and the message of the alert box. Embedded newline characters (`'\n'`) produce line breaks.

In combination with [fl_add_timeout()], page 300, it is easy to develop a timed alert routine that goes away when the user pushes the OK button or when a certain time has elapsed:

```
static void dismiss_alert(int ID, void *data) {
    fl_hide_alert();
}

void show_timed_alert(const char *s1, const char *s2,
                      const char *s3, int centered) {
    fl_add_timeout( 10000, dismiss_alert, 0 ); /* ten seconds */

    /* fl_show_alert blocks, and returns only when the OK button
       is pushed or when the timeout, in this case, 10 seconds,
       has elapsed */

    fl_show_alert(s1, s2, s3, centered);
}
```

Then you can use `show_timed_alert()` just as `fl_show_alert()` but with added functionality that the alert will remove itself after 10 seconds even if the user does not push the OK button.

```
int fl_show_question(const char *message, int def);
void fl_hide_question(void);
```

Again shows a message (with possible embedded newlines in it) but this time with a Yes and a No button. `def` controls which button the mouse pointer should be on: 1 for Yes, 0 for No and any other value causes the form to be shown so the mouse pointer is at the center of the form. 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.

If the question goodie is removed programmatically via [fl_hide_question()], page 70, the default `def` as given in [fl_show_question()], page 70 is taken. If no default is set, 0 is returned by [fl_show_question()], page 70. The following code segment shows one way of using [fl_hide_question()], page 70

```
void timeout_yesno(int id, void *data) {
    fl_hide_question();
}
```

```
...

fl_add_timeout(5000, timeout_yesno, 0);

/* show_question blocks until either timeouts or
   one of the buttons is pushed */

if (fl_show_question("Want to Quit ?", 1))
    exit(0);

/* no is selected, continue */

...  /* rest of the code *.
```

In the above example, the user is given 5 seconds to think if he wants to quit. If within the 5 seconds he can't decide what to do, the timeout is triggered and [fl_show_question()], page 70 returns 1. If, on the other hand, he pushes the No button before the timeout triggers, [fl_show_question()], page 70 returns normally and [fl_hide_question()], page 70 becomes a no-op.

```
int fl_show_choice(const char *s1, const char *s2, const char *s3,
                   int numb, const char *b1, const char *b2,
                   const char *b3, int def);

int fl_show_choices(const char *s, int numb,
                    const char *b1, const char *b2, const char *b3,
                    int def);

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

void fl_hide_choice(void);
```

The first routine shows a message (up to three lines) with one, two or three buttons. numb indicates the number of buttons. b1, b2 and b3 are the labels of the buttons. def can be 1, 2 or 3, indicating the default choice. The second routine is similar to the first except that the message is passed as a single string with possible embedded newlines in it. Both routines return 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. If the choice goodie is removed by [fl_hide_choice()], page 71, the default def is returned.

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);
void fl_hide_input(void);
```

This shows a box with one line of message (indicated by `str1`), and an input field into which the user can enter a string. `defstr` is the default input string placed in the input box. In addition, three buttons, labeled `Cancel`, `OK` and `Clear` respectively, are added. The button labeled `Clear` deletes the string in the input field. The routine returns the string in the input field when the user presses the `OK` button or the `<Return>` key. The function also returns when button `Cancel` is pressed. In this case, instead of returning the text in the input field, `NULL` is returned. This routine can be used to have the user provide all kinds of textual input.

Removing the input field programmatically by calling `[fl_hide_input()], page 71` results in `NULL` being returned by `[fl_show_input()], page 71`, i.e., it's equivalent to pressing the `Cancel` button.

A similar but simpler routine can also be used to obtain textual input

```
const char *fl_show_simple_input(const char *str1, const char *defstr);
```

The form shown in this case only has the `OK` button. The example program `goodies.c` shows you these goodies.

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

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

To, for example, change the label of the `Dismiss` button to `"Go"` in the alert form, code similar to the following can be used after calling `[fl_initialize()], page 277` but before any use of the alert goodie:

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

Currently the following goodies resources are supported:

`flAlert.title`
: The window title of the alert goodie

`flAlert.dismiss.label`
: The label of the `Dismiss` button

`flQuestion.yes.label`
: The label of the `Yes` button

`flQuestion.no.label`
: The label of the `No` button

`flQuestion.title`
: The window title of the Question goodie

`flChoice.title`
: The window title of the Choice goodie

`*.ok.label`
: The label of the `OK` button

Note that all goodies are shown with `FL_TRANSIENT` and not all window managers decorate such forms with titles. Thus the title setting in the above listing may not apply.

6.2 Command Log

In a number of situations, a GUI is created specifically to make an existing command-line oriented program easier to use. For stylistic considerations, you probably don't want to have the output (`stderr` and `stdout`) as a result of running the command printed on the terminal. Rather you want to log all the messages to a browser so the user can decide if and when to view the log. For this, a goodie is available

```
long fl_exe_command(const char *cmd, int block);
```

This function, similar to a `system(3)` call, forks a new process that runs the command `cmd`, which must be a (null-terminated) string containing a command line passed to the (sh) shell. The output (both `stderr` and `stdout`) of `cmd` is logged into a browser, which can be presented to the user when appropriate (see below). The `block` argument is a flag indicating if the function should wait for the child process to finish. If the argument `block` is true (non-zero), the function waits until the command `cmd` completes and then returns the exit status of the command `cmd` (i.e., the status one gets form `wait()` or `waitpid()`, so use `WEXITSTATUS()` on it if you want the return or `exit()` value from the program started)). If the argument `block` is false (0), the function returns immediately without waiting for the command to finish. In this case, the function returns the process ID of the child process or -1 if an error occured.

Unlike other goodies, `[fl_exe_command()], page 73` does not deactivate other forms even in blockng mode. This means that the user can interact with the GUI while `[fl_exe_command()], page 73` waits for the child process to finish. If this is not desired, you can use `[fl_deactivate_all_forms()], page 296` and `[fl_activate_all_forms()], page 296` to wrap the function.

If `[fl_exe_command()], page 73` is called in non-blocking mode, the following function should be called to clean up related processes and resources before the caller exits (otherwise a zombie process may result)

```
int fl_end_command(long pid);
```

where `pid` is the process ID returned by `[fl_exe_command()], page 73`. The function suspends the current process and waits until the child process is completed, then it returns the exit status of the child process or -1 if an error has occurred.

There is another routine that will wait for all the child processes initiated by `[fl_exe_command()], page 73` to complete

```
int fl_end_all_command(void)
```

The function returns the status of the last child process.

You can also poll the status of a child process using the following routine

```
int fl_check_command(long pid);
```

where `pid` is the process ID returned by `[fl_exe_command()], page 73`. The function returns the following values: 0 if the child process is finished; 1 if the child process still exists (running or stopped) and -1 if an error has occurred inside the function.

If some interaction with the command being executed is desired, the following functions may be more appropriate. These functions operates almost exactly as the `popen(3)` and `pclose(3)` functions:

```
FILE *fl_popen(const char *command, const char *type);
```

```
int fl_pclose(FILE *stream);
```

The [fl_popen()], page 73 function executes the command in a child process, and logs the stderr messages into the command log. Further, if type is "w", stdout will also be logged into the command browser. [fl_pclose()], page 73 should be used to clean up the child process.

To show or hide the logs of the command output, use the following functions

```
int fl_show_command_log(int border);
void fl_hide_command_log(void);
```

where border is the same as that used in [fl_show_form()], page 292. These two routines can be called anytime anywhere after [fl_initialize()], page 277 has been invoked.

The command log is by default placed at the top-right corner of the screen. To change the default placement, use the following routine

```
void fl_set_command_log_position(int x, int y);
```

where x and y are the coordinates of the upper-left corner of the form relative to the root window. The logging of the output is accumulative, i.e., [fl_exe_command()], page 73 does not clear the browser. To clear the browser, use the following routine

```
void fl_clear_command_log(void);
```

It is possible to add arbitrary text to the command browser via the following routine

```
void fl_addto_command_log(const char *txt);
void fl_addto_command_log_f(const char *fmt, ...);
```

where txt for fl_addto_command_log() is a string and fmt for fl_addto_command_log_f() is a format string like for printf() that gets expanded using the following arguments. This string, with possible embedded newlines, gets added to the last line of the browser using [fl_addto_browser_chars()], page 171.

Finally, there is a routine that can be used to obtain the GUI structure of the command browser

```
typedef struct {
    FL_FORM   * form;           /* the form */
    FL_OBJECT * browser;        /* the browser */
    FL_OBJECT * close_browser;  /* the close button */
    FL_OBJECT * clear_browser;  /* the clear button */
} FD_CMDLOG;

FD_CMDLOG *fl_get_command_log_fdstruct(void);
```

From the information returned the application program can change various attributes of the command browser and its associated objects. Note however, that you should not hide/show the form or free any members of the structure.

6.3 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. Once the user presses the mouse one of the

entries 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).

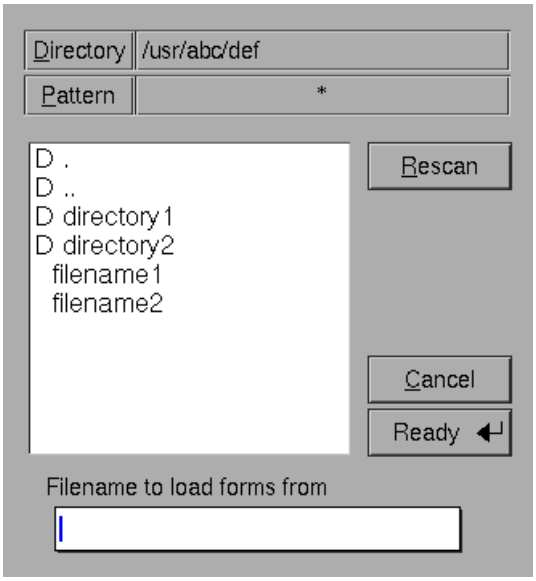
6.4 File Selector

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 all files in directory `directory` are listed that satisfy the pattern `pattern` (see Fig 6.1). `pattern` can be any kind of regular expression, e.g., `[a-f]*.c`, which would list 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. The user can choose a file from the list given and the function then 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 clicking on the box with the name of the currently displayed directory to edit it manually, or by double-clicking on the name of a directory (shown with a 'D' in front of it) shown in the list. If the directory content changes while it is being displayed in the file selector the `ReScan` button can be used to request a rescan of the directory.



In a typical application, once the file selector goodie is shown, it is up to the user when the file selector should be dismissed by pushing `Ready` or `Cancel` button. In some situations the application may want to remove the file selector on it's own. To this end, the following routine is available

```
void fl_hide_fselector(void);
```

The effect of removing the file selector programmatically is the same as pushing the `Cancel` button. 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`.

To change the font the file selector uses, the following routine can be used:

```
void fl_set_fselector_fontsize(int font_size);
void fl_set_fselector_fontstyle(int font_style);
```

These routines change the font for all the objects on the form. It is possible to change the font for some of the objects (e.g., browser only) using `[fl_get_fselector_fdstruct()], page 78` explained later.

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

```
void fl_invalidate_fselector_cache(void);
```

before `[fl_show_fselector()], page 75`. 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:

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

A false (0) parameter (re)enables directory caching.

The user can also change the pattern by clicking the mouse on top of it 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(int (*callback)(const char *, void *),
                               void *user_data);
```

where the second argument of the callback is the `user data`. The return value of the callback function is currently not used. 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.

Please note that when a file selector has a callback installed the field for manually entering a file name isn't shown.

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()], page 292`. 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 border);
```

The `border` request by this function is the same as in `[fl_show_form()], page 292`, but `FL_NOBORDER` is ignored.

If the arguments `directory`, `pattern` or `default` passed to `[fl_show_form()], page 292` are empty strings or `NULL`, the previous value is used (with some reasonable defaults getting used when this happens the first time). Thus the file selector "remembers" all the settings the selector had 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);
```

It is also possible to programatically set new values for the default directory and pattern by using the functions

```
int fl_set_directory( const char * dir );
void fl_set_pattern( const char * pattern );
```

`[fl_set_directory()], page 77` returns 0 on success and 1 on failure, either because the argument was a `NULL` pointer or not a valid directory.

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);
```

The argument `data` is passed to the callback. Whenever this application specific button is pushed, the callback function is invoked.

To remove an application specific button, use the following routine

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

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.

If, for whatever reasons, there is a need to get the fselector's form 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 of a fselector's form maybe necessary. To this end, the following routine exists

```
typedef struct {
    FL_FORM   * fselect;
    void      * vdata;
    char      * cdata;
    long        ldata;
    FL_OBJECT * browser,
              * input,
              * prompt,
              * resbutt;
    FL_OBJECT * patbutt,
              * dirbutt,
              * cancel,
              * ready;
    FL_OBJECT * dirlabel,
              * patlabel;
    FL_OBJECT * appbutt[3];
} FD_FSELECTOR;

FD_FSELECTOR *fl_get_fselector_fdstruct(void);
```

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

```
FD_FSELECTOR *fs = fl_get_fselector_fdstruct();

fl_set_object_label(fs->ready, "Go !");
fl_fit_object_label(fs->ready, 1, 1);
```

Since the return value of `[fl_get_fselector_fdstruct()], page 78` is a pointer to an internal structures, the members of this structure should not be modified.

In the listing of files in a directory special files are marked with a prefix in the browser (for example, `D` for directories, `p` for pipes 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);
```

where `dir` is the marker character for directories, `fifo` the marker for pipes and FIFOs, `socket` the marker for sockets, `cdev` the marker for character device files and, finally, `bdev` the marker character for block device files.

Although file systems under Unix are similar, they are not identical. In the implementation of the file selector, the subtle differences in directory structures 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 specified by `pattern` (not exactly true, see below). The function returns the address of an array of type `FL_Dirlist` with `nfiles` if successful and `NULL` otherwise. By default, directory entries are cached. Passing the function a true (non-zero) value for the `rescan` argument requests a re-read.

`FL_Dirlist` is a structure defined as follows

```
typedef struct {
    char          * name;        /* file name */
    int             type;        /* file type */
    long            dl_mtime;    /* file modification time */
    unsigned long   dl_size;     /* file size in bytes */
} 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 or FIFO

`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()], page 79`, use the following call

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

Note that a cast may be required to get rid of compiler warnings due to the `const` qualifier of the return value of `[fl_get_dirlist()], page 79`. See demo program `dirlist.c` for an example use of `[fl_get_dirlist()], page 79`.

Per default not all types of files are returned by `[fl_get_dirlist()], page 79`. The specific rules for which types of file are returned are controlled by an additional filter after the pattern filter. It has the type

```
int default_filter(const char *name, int type);
```

and is called for each entry found in the directory that matched the pattern. This filter function should return true (non-zero) if the entry is to be included in the directory list. The default filter is similar to the following

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

i.e., per default only directories, normal files and symbolic links are shown (the first argument of the function, the file name, isn't used by the default filter).

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);
```

As noted before, directories are by default not subject to filtering. If, for any reason, it is desirable to filter also directories, use the following routine with a true flag

```
int fl_set_dirlist_filterdir(int flag);
```

The function returns the old setting. Since there is only one filter active at any time in XForms, changing the filter affects all subsequent uses of file browsers.

By default, the files returned are sorted alphabetically. You can change the default sorting using the following routine:

```
int fl_set_dirlist_sort(int method);
```

where `method` can be one of the following

`FL_NONE` Don't sort the entries

`FL_ALPHASORT`
: Sort the entries in alphabetic order - this is the default

`FL_RALPHASORT`
: Sort the entries in reverse alphabetic order

`FL_MTIMESORT`
: Sort the entries according to the modification time

`FL_RMTIMESORT`
: Sort the entries according to the modification time, but reverse the order, i.e., latest first.

`FL_SIZESORT`
: Sort the entries in increasing size order

`FL_RSIZESORT`
: Sort the entries in decreasing size order

`FL_CASEALPHASORT`
: Sort the entries in alphabetic order with no regard to case

`FL_RCASEALPHASORT`
: Sort the entries in reverse alphabetic order with no regard to case.

The function returns the old sort method. For directories having large numbers of files, reading the directory can take quite a long time due to sorting and filtering. Electing not to sort and (to a lesser degree) not to filter the directory entries (by setting the filter to `NULL`) can speed up the directory reading considerably.

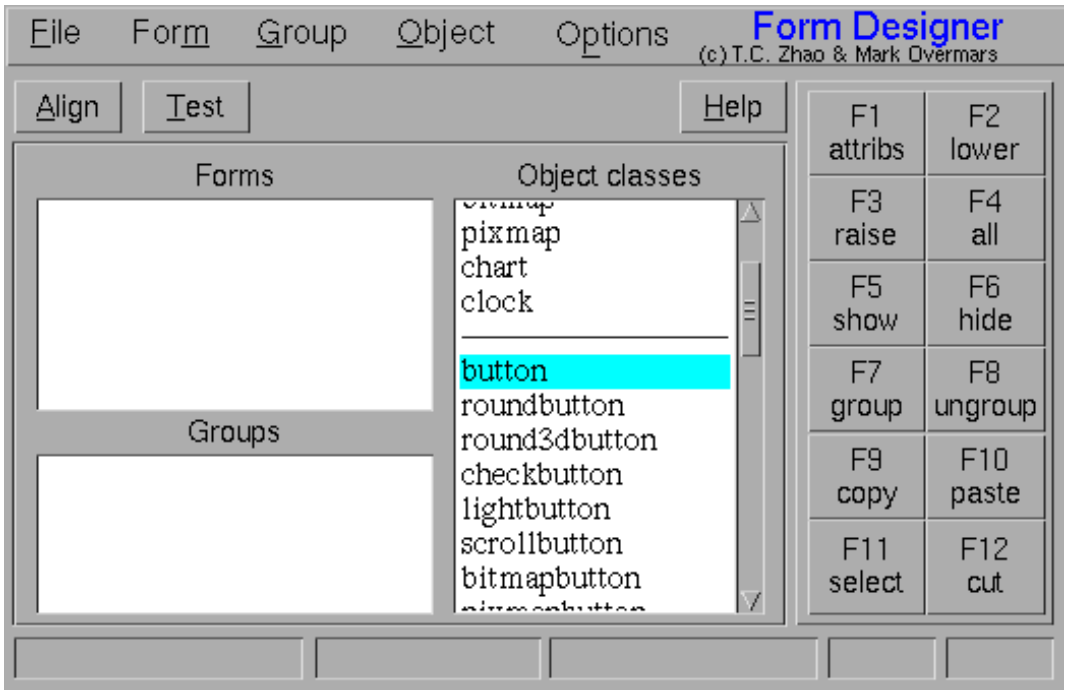
Part II - The Form Designer

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 current version is mostly a layout tool and not a programming environment, not yet anyway. This means that the Form Designer does not allow you to initialize all your objects. You can, however, initialize some objects, e.g., you can set the bounds of a slider inside the Form Designer. Eventually full support of object initialization will be implemented.

8 Getting Started

To start up the Form Designer simply type fdesign without any arguments. (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. 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.

Directly below the menu you have a row of buttons for alignment, testing and getting help, see below.

Then there's a panel with three browsers. At the left there is a list of all existing forms. When the program is started without an already existing file as an argument the list is empty, indicating that there are no forms yet. There's no upper limit to the number of forms that can be created but you can only work on exactly one form at a time. Use this list to switch between the different forms. Below the forms list is another list showing all groups in the form you're currently working on. It will be empty for a new form because there are no groups yet. Ignore this at the moment as we will come back to groups and their uses later.

Just right of those two lists you find a list of all the different classes of objects that can be placed into the form. Use the mouse to select the class of a new object you want to add to the form.

On the right side beside the panel with the browsers 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 corresponding function keys on the

keyboard (while the keyboard focus is on the window with the form). The functions of these keys will be described below.

To create a new form select the "New Form" entry in the "Form" menu. A little popup box will appear, prompting you for the name of the new form. This is the name under which the program you're going to write will know the form. Thus you will have to provide a name which must be a legal C variable name. Type in the name and press `<Ok>`. Now the color of the window for showing the form you're working on changes to that of the default background color of forms. (Actually, each new form gets created with a box already covering its entire area, what you see is the color of this box. You can change most properties of this box using the methods described below. Just its size is fixed to the size of the form, which can be simply changed by resizing the window.) Note that the form's name is added to the list of forms in the control panel.

To add an object to the form select its class in the control panel by selecting an item the list of object classes. Then move the mouse into the window with the form you are working on and drag the mouse while pressing the left mouse button. By keeping the mouse button pressed you create a box that has the size of the object to be created. Release the button and the object will appear. Note that a red outline appears around the new object. This indicates that the object is selected. In this way you can put all kinds of objects on the form.

Object already created can be modified in several ways. You can move them around, change their sizes or their attributes. To this end first select the object by left-clicking on it. But this only works if there isn't an obkect class selected in the object class browser in the control panel. To get rid of such a selection either click on the selected entry in this browser or by right-click somewhere in the window with the new form. When the object is selected a red outline appears around it. You now will be able to drag the object around with the mouse. By grabbing the object at one of the four red corners you can change its size. It is also possible to select multiple objects and move or scale them simultaneously. See below for details.

To change the object's attributes, e.g., its label, simply double-click on it with the left mouse button. Or single-click on it and then press the function key `<F1>` (or click on the button labeled "F1 attribs" in the control panel or select "Object attributes" from the "Object" menu). A new form appears in which you can change all the different attributes. Their meanings should be clear (if you have read the documentation on the Forms Library). Change the attributes you want to change and finally press the button labeled "Accept". To reset all attributes to their original values press "Restore" (or "Cancel" if you also want to close the window for modifying the 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 have more than one form. 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 select "Save" or "Save As"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 routines with names like `create_form_xxx()` (replace `xxx` with the names you gave to the forms) to create the different forms you designed.

These are the basic ideas behind the Form Designer. In the following chapters we describe the program in more detail.

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 the `-geometry` option is not given. 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 it to simply read a list of fdesign output files (the `.fd` files) and emit the corresponding C-routines and header files. This can be useful e.g., in automatically compiling packages in Makefiles. Note that the input `.fd` will only be read but never modified when this option is used.

`-migrate` *fd-file-list*
: When fdesign is invoked with the `-convert` option it just creates new `.c` and `.h` files but leaves the `.fd` files unmodified. In some situations, e.g., if you also want to automatically upgrade `.fd` files created with older versions of fdesign, you can instead use the `-migrate` option which does all what the `-convert` option does but also writes out a new version of the `.fd` file it just read in. It also does a few extra checks, e.g., it will test if XBM and XPM image files used for bitmaps and pixmaps actually exist (if they don't the newly generated `.fd` file won't reference them anymore, so carefully look out for error messages and, if necessary, restore it from the generated `.fd.bak` backup file).

`-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 some types of windows so that you can move them (only necessary with some window managers).

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

`-nocode`
: Suppresses the output of UI code. Sometimes useful if the UI code is not to be generated interactively, but rather generated by the make process using "fdesign -convert".

`-I` *`header`* Changes the output include file from `<forms.h>` to header. Per default, the header file name will be enclosed in angle brackets ('`<`' and '`>`') unless the name of the include file specified is already enclosed in double quote ('`"`'). Useful on systems where `forms.h` is renamed to something else or if you need an application header file with e.g., definitions of constants/defines for the UI that itself includes the `forms.h` file.

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

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

`-lax` Suppresses checking of variable and callback function names for being aceptable C variable names

`-bw` *`borderwidth`*
Changes the 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`	string	Geometry
`control.border`	bool	XForms borderwidth
`control.geometry`	string (position only)	Control window geometry
`attributes.geometry`	string (position only)	Attributes window Geometry
`attributes.background`	string (e.g., gray80)	Attributes window background
`align.geometry`	string (position only)	Align window geometry
`help.geometry`	string (position only)	Help window geometry
`convert`	bool	Convert
`unit`	string	Unit
`altformat`	bool	AltFormat
`xformHeader`	string	Header file name
`helpFontSize`	int	Help font size
`main`	bool	Main

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

10 Creating Forms

10.1 Creating and Changing Forms

To create a new form use the "New Form" entry in the "Form" menu at the top. When asked for the new form's name enter a (unique) name that is a valid C identifier. 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. The easiest way is to simply change the size of the main window displaying the form and the form will resize itself to fit the new size. Otherwise you can use the "Resize Form" entry in the "Form" menu, in which case you can enter the width and height of the form manually.

To change the name of the current visible form use the "Rename Form" entry in the "Form" menu. You will be prompted for the new form name.

To delete a form use the "Delete Form" entry in the "Form" menu . The current form will be removed after a box asking you if you're sure had been shown.

10.2 Adding Objects

To add an object choose the class of the new object from the list of object classes in the middle of the control panel. Next drag the left mouse button within the main form. A rubber box outlining the size of the new 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 per default. How to change the default is described below in the context of alignments.

10.3 Selecting Objects

To perform operations on objects that are already visible in the form, we first have to select them. Any mouse button can be used for selecting objects. Simply single-click on the object you want to select. A red outline will appear, indicating that the object is selected. Another way of selecting objects is to use the `<Tab>` or `<F11>` keys or the button labeled `F11`, all of which iterates over the object list and selects the next object upon each press (the only object not selected this way is the backface object).

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

To add objects to an already existing selection, hold down the `<Shift>` key and press the left mouse button inside the object. 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 object) 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 changing its attributes is allowed.

10.4 Moving and Scaling

To move an object (or a collection of objects) to a new place, first select it (them) as described above. Next press the left mouse button inside the bounding box (not too near to 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 by dragging the mouse.

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

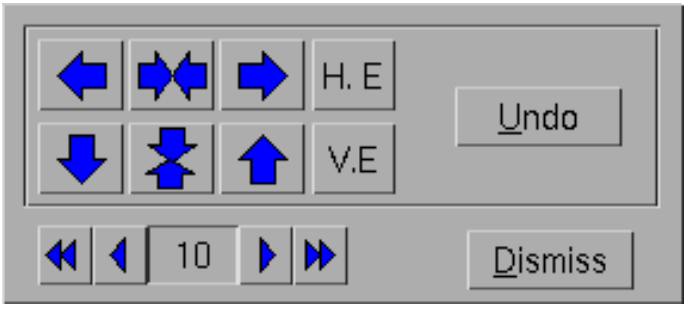
For precise object movement the cursor keys can be used. Each press of the four directional cursors keys moves the selected object by 10 pixels per default. To change the step sizepress one of the numbers from 0 to 9 with 0 indicating 10 pixels.

If the `<Shift>` key is kept pressed down instead of moving the object its 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 labeled "Dismiss" on the form or by clicking the "Align" button again.

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



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

In the alignment form you can also indicate the "snapping size" when moving or resizing objects, using the counter at the bottom. Default snapping is 10 pixels. Snapping helps in making objects of the same size and aligning them nicely.

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 hiding the buttons (if you put a framebox over a button, the button will be visible but appears to be inactive!). This is definitely not what you want. The Form Designer makes it possible to raise objects (bring them to the top) or lower them (put them at the bottom). So you can lower e.g., a box to move it under some buttons. Raising or lowering objects is very simple. First select the objects and next press the function key `<F2>` to lower the selection or `<F3>` to raise it.

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

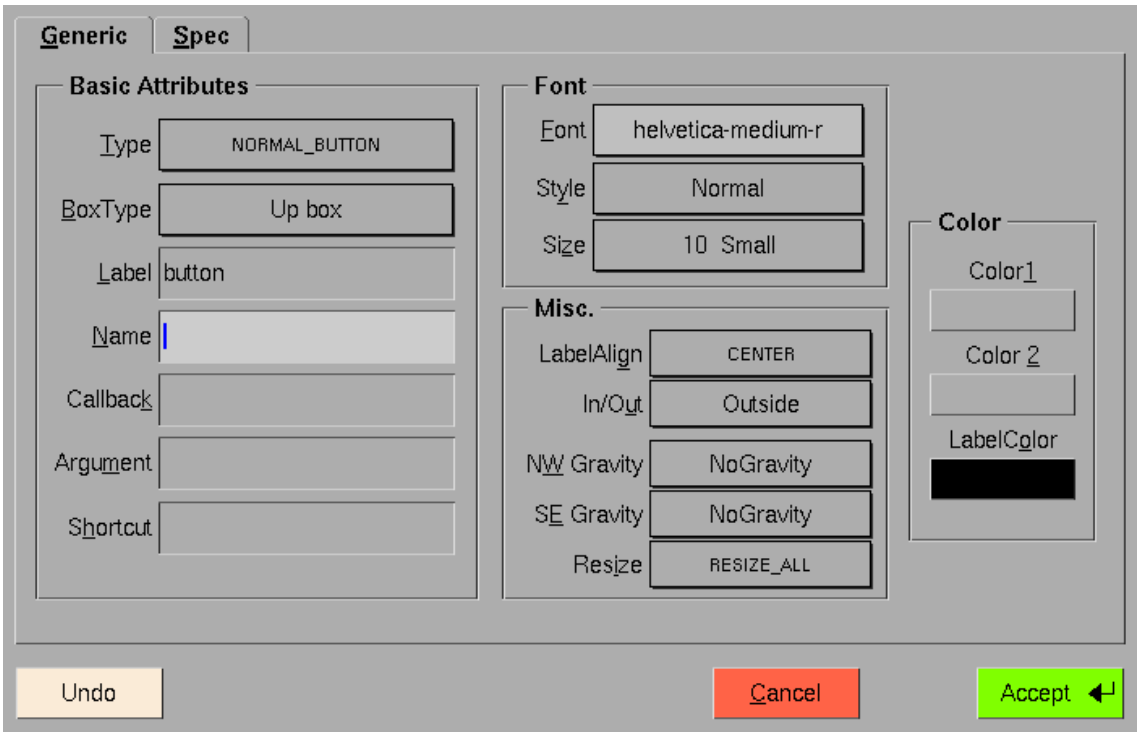
10.7 Setting Attributes

To set attributes like type, color, label, etc., of an object first select it (using the left mouse button) and next press the function key `<F1>` (or click on the button labeled "F1"). Also a double click (with the left mouse button) selects the object and opens up the form for changing the objects attributes. 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 object class. In this case you cannot change the labels, names, etc. because you probably want them to remain different for the different objects.

The form for changing object attributes allows you to modify all the different settings. Before we continue, the organization of the attributes form and classification of attributes needs a little explanation. Attributes of an object are divided into two categories. The generic attributes are shared by all objects. These include type, colors, label, callback function etc. The other class of attributes are those that are specific to a particular object class, such as slider bounds, precision etc. When the attribute form is first shown, only the generic attributes are shown. Press on the tab rider "Spec" to get to a second form for the object class specific attributes (press the tab rider "Generic" to switch back to the generic attributes part).

10.8 Generic Attributes

The form for setting generic attributes contains four fields for setting different groups of generic properties, discussed in the following. Once you are satisfied with the settings, press the button labeled "Accept" and the form will disappear. If you don't want to change the attributes after all press the button labeled "Cancel". You may also reset the values to what they were when you started editing them by clicking on the "Undo" button.



10.8.1 Basic Attributes

The basic attributes include the type, boxtype, name, label string, the callback function with its arguments associated with the object and a shortcut.

For most object classes several different types exist. The type of the object under consideration can be selected via a choice object.

Most objects can also be drawn using different boxtypes. Normally, the default should do, but using the choice object labeled "BoxType" you can switch to a different box type (but note that not all choices may result in a different way the object is drawn and some may look rather ugly).

Nearly all objects have a label that can be drawn at different positions within or outside of the object. The input field labeled "Label" lets you set the label string (it may also include return characters, i.e., `\n`, for line breaks).

An object may have a name by which it can be accessed within the program. This name must be a valid C (or simple C++) variable identifier and can be set via the input field labeled "Name". You need to make sure that there are no objects with the same name!

If instead of having e.g., the function `[fl_do_forms()], page 296` return when an object is triggered a callback may be invoked instead. You can set the name of the callback function in the input field labeled "Callback". Obviously, this must be a valid C or C++ function name. When a callback function is set you must also specify the argument passed to the callback function together with the object's address via the input field labeled "Argument". This normally will be a (long) integer (defaulting to 0 if not specified). Using this value it is e.g., possible to distinguish between different objects when all use the same callback function.

10.8.2 Font

In the field labeled "Font" you can set properties of the font to be used for the label of the object. You can select between different types of fonts, the style the label is drawn in (normal, shadowed, engraved or embossed) and the size of the font to be used. All three types of properties can be selected via choice objects.

10.8.3 Misc. Attributes

The field labeled "Misc. Attributes" allows the setting of a number of attributes that don't fit into any other category.

First you can set the alignment of the object's label. It can be placed inside the object or outside of it and in 9 different positions. Use the choice objects labeled "Label/Align" and "In/Out" for this purpose.

Another important property of an object is how it reacts if the size of the form it belongs to is changed. It may keep its original size or may be resized in x- or y-direction or both. The details are controlled via its resize and gravity properties as described in chapter 4.

With the choice objects labeled "Resize" you can control if an object is to be resized in x- or y-direction or both or none. You may also specify if the object's upper left hand corner or its lower right hand corner is supposed to keep a fixed distance from the form's borders via the choice objects labeled "NW Gravity" and "SE Gravity". Please note that these properties aren't orthogonal, with the NWGravity and SEGravity overriding the resize property if necessary (also see the program `grav` in the `demo` directory that lets you experiment with these properties).

10.8.4 Colors

Within the "Color" field you can set three colors for the object. The colors of the object itself are controlled via the buttons labeled "Color 1" and "Color 2", while the button labeled "LabelColor" is for setting the color the label is drawn in.

Clicking on any of the three buttons will result in a new form being shown in which you can select one of the predefined colors from the internal colormap. You also can select one of the "free" colors but since these colors aren't set yet they will appear as black in the form for selecting the color.

While it's rather obvious what the label color is, the meaning of "Color 1" and "Color 2" varies a bit with the class and type of the object. E.g., for (normal buttons the first color is the normal color of the button while the second one is the color it's drawn in while the button is pressed, while for a browser that allows selection the first color is the background color and the second color is the color selected lines are highlighted with. Since there are

too many combinations of object classes and types to be discussed here comprehensively please refer to a following chapter where the exact properties of all objects are described in detail.

10.9 Object Specific Attributes

Many objects have attributes that are specific to its object class, such as slider bounds, precision etc. You can access these attributes (if existent) via the tab rider labeled "Spec". In most cases the meaning of these attributes hopefully is self-explanatory (otherwise see the detailed description of the different object classes in Part III) and all changes made are shown immediately so you can see what effects the changes have on the object. Once satisfactory results have been achieved the press button labeled "Accept" to accept the settings (clicking on the tab rider "Generic" has the same effect). Two additional buttons, "Cancel" and "Restore", are available to cancel the changes (and quit the attribute settings form) and restore the defaults, respectively.

One particular aspect of the pixmap/bitmap button initialization needs a little more explanation as the setting of button labeled "Use data" has no effect on the appearance of the button in fdesign but nonetheless affects the generated code. By default, the "Use data" button is off, indicating the pixmap/bitmap file specified is to be loaded dynamically at run time via `[fl_set_pixmapbutton_file()], page 125` (or the bitmap counterpart function). If "Use data" is on, the specified file and its associated data will be `#include`'d at compile time so the data becomes part of the code. Depending on the application setup, you may choose one method over the other. In general, including the data in the code will make the code slightly larger, but it avoids problems with finding the specified file at runtime. The button labeled "Full Path" only applies if "Use Data" is on. If "Full Path" is also on, the pixmap file will be `#include`'d using the full path, otherwise only the filename is used, presumably the compile process will take care of the path via the `-I` flag in some system dependent way.

10.10 Cut, Copy and Paste

You can remove objects from the form by first selecting them and then pressing the `<F12>` function key or simply by double-clicking on it with the right mouse button. The object(s) will disappear but in fact will be saved in a buffer. You can put it back into the form (or in another form) by pasting, 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 (i.e., without removing the original object(s)) into the buffer using `<F9>`. The content of the bufer can now be put into the same or another form. This allows for a simple mechanism of making multiple copies of a set of objects and for moving information from one form to another.

To clone the currently selected object, hold down the `<Shift>` key and then drag the selected object to a new position. The cloned object will have exactly the same attributes as the original object except for its name and shortcut keys (would these also be cloned, the generated code would not be compilable or cause runtime misbehavior).

When you copy objects belonging to a common group just the individual objects of the group will be copied, but they won't belong to a common group anymore.

10.11 Groups

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

In the control panel there is a list of groups in the current form. As long as you don't have groups, this list will be empty. To create a group, select the objects that should become members of the group and press the function key `<F7>`. You will be prompted for the name of the group. This should be a legal C variable name (under which the group will be known to the application program) or should be left empty. This name will be added to the list. In this way you can create many groups. Note that each object can be member of 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. A simple click on a groups name will select this group and deselect all objects not belonging to it. Clicking on a groups name while the `<Shift>` key is pressed down adds the group to the cuurent selection.

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.

You can use the menu `"Rename group"` from the `"Group"` menu to change the name of a selected group. Only a single group may be selected when changing the name.

10.12 Hiding and Showing Objects

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 then press `<F5>`. A possibly better way is 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 and then 'unhide' them.

10.13 Testing Forms

To test the current form, press the button labeled `"Test"`. The form will be displayed in the center of the screen anf a panel appears at the top right corner of the screen. This panel shows you the objects returned and callback routines invoked when working with the form. In this way you can verify whether the form behaves correctly and whether all objects have

either callback routines or names (or both) associated with them. You can also resize the form (if the backface of the form allows resizing) to test the gravity and resizing behaviour. You can play with the form as long as you want. When ready, press the "Stop Testing" button.

Note that any changes you made to the form while testing (including its size) 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's preset value.

11 Saving and Loading Forms

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

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

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

There are two different kind of formats for the C-code generated. The default format allows more than one instance of the form created and uses no global variables. The other format, activated by the `altformat` option given on the command line or switched on via 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 that can be changed (see below).

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

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

```
#ifndef FD_foobar_h_
#define FD_foobar_h_

/* call back routines if any */

extern void callback(FL_OBJECT *, long);

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

typedef struct {
    FL_FORM *   bar;
    void *      vdata;
```

```
    char *      cdata;
    long        ldata;
    FL_OBJECT * f2obj1;
    FL_OBJECT * f2obj2;
} FD_bar;

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

#endif /* FD_foobar_h */
```

and the corresponding C file:

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

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

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

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

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

    fdui->bar = fl_bgn_form(....);
    fdui->f2obj1 = fl_add_cccc(....);
    fdui->f2obj2 = fl_add_dddd(....);
    fl_end_form();

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

The application program would look something like the following:

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

/* add call back routines here */

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

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

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

    fl_show_form(fd_foo->foo, ...);

    /* rest of the program */
}
```

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

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

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

```
fd_foo = create_form_foo();
...
```

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

```
void callback(FL_OBJECT *obj, long data) {
    FD_foo *fd_foo = obj->form->fdui;
    fl_set_object_dddd(fd_foo->f1obj2, ....);
}
```

Of course this setup still leaves the problems accessing objects on other forms unsolved although you can manually set the `form->u_vdata` to the other `FD_` structure:

```
fd_foo->form->u_vdata = fd_bar;
```

or use the `vdata` field in the `FD_` structure itself:

```
fd_foo->vdata = fd_bar;
```

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

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

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

```
void callback(FL_OBJECT *obj, long data) {
    FD_foo *fd_foo = obj->form->u_vdata;
    fl_set_object_dddd(fd_foo->f1obj2, ....);
}
```

Not pretty, but adequate for practical purposes. Note that the `FD_` structure always has a pointer to the form as the first member, followed by `vdata`, `cdata` and `ldata`. There's also a `typedef` for a structure of type `FD_Any` in `forms.h`:

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

you can use a cast to a specific `FD_` structure to get at `vdata` etc. Another alternative is to use the `FD_` structure as the user data in the callback[1]

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

and use the callback as follows

```
void callback(FL_OBJECT *obj, long arg) {
    FD_foo *fd_foo = (FD_foo *) arg;
    fl_set_object_lcolor(fd + foo->f1obj1, FL_RED);
    ...
}
```

Avoiding globals is, in general, a good idea, but as everything else, also an excess of a good things 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 object, say, `f1obj2` via `[fl_set_button()], page 124` or `[fl_set_input()], page 152`. Now the state of `f1obj2` is changed and it needs to be handled. You probably don't want to put much code for handling `f1obj2` in `f1obj1`'s callback. In this situation, the following function is handy

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

`fl_call_object_callback(fdfoo->f1obj2)` will invoke the callback for `f1obj2` callback in exactly the same way the main loop would do 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;
```

[1] Unfortunately, this scheme isn't legal C as a pointer may be longer than a long, but in practice, it should work out ok on virtually all platforms.

```
 extern FL_OBJECT *f1obj1,
                  *f1obj2,
                  ...;
extern FL_OBJECT *f2obj1,
                 *f2obj2,
                 ...;

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

The C-routines:

```
FL_FORM *foo,
        *bar;

FL_OBJECT *f1obj1,
          *f1obj2,
          ...;
FL_OBJECT *f2obj1,
          *f2obj2,
          ...;

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"

/* Here go the callback routines */
....
```

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

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" item 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.

12 Language Filters

Please note: This chapter is probably completely outdated!

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

`ada95` by G. Vincent Castellano `gvc@ocsystems.com`

`perl` by Martin Bartlett `martin@nitram.demon.co.uk`

`Fortran` by G. Groten `zdv017@zam212.zam.kfa-juelich.de` and Anke Haeming `A.Haeming@kfa-juelich.de`

`pascal` by Michael Van Canneyt `michael@tfdec1.fys.kuleuven.ac.be`

`scm/guile` by Johannes Leveling `Johannes.Leveling@Informatik.Uni-Oldenburg.DE`

`python` by Roberto Alsina `ralsina@ultra7.unl.edu.ar`. It would appear that author of python binding is no longer working on it.

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

12.1 External Filters

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

By default, Form Designer only outputs the `.fd` and C files. If the presence of `-ada`, `-perl`, `-python`, `-fortran`, `-pascal` or `-scm` command line options to Form Designer is detected, then after emitting the default output, Form Designer invokes the the external filter with the root filename (without the `.fd` extension) as an argument, together with possible other flags, to the filter. Any runtime error messages are presented to the user in a browser. The filter name by default is `fd2xxxx` where `xxxx` is the language name (such as `fd2perl` etc.), which can be changed using the `-filter` command line option (or equivalent resources).

The resources that are relevant to the filter are listed below

Resource	Type	Default
language	string	C
filter	string	None

12.2 Command Line Arguments of the Filter

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

`-callback`
: callback stubs are generated

`-main`
: main stub is generated

`-altformat`
: output in alternate format

`-compensate`
: emit size compensation code

13 Generating Hardcopies

A variety of tools are available that can be used to turn your carefully constructed (and hopefully pleasing) user interfaces into printed hardcopies or something appropriate for inclusion in your program document. Most of these involves saving a snapshot of your interface on the screen into a file. Then this file is translated into something that a printer can understand, such as `PostScript`.

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

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

The translation can be done in two ways. One way is to simply give the Form Designer the command line option `-ps` to have it output `PostScript` directly. or you can run `fd2ps` stand alone using the command

```
fd2ps fdfile
```

where `fdfile` is the Form Designer output with or without the `.fd` extension. The output is written into a file named `fdfile.ps`.

`fd2ps` accepts the following command line options when run as a stand-alone program

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

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

`-l` This option requests landscape orientation.

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

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

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

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

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

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

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

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

Letter	8.5 x 11 inch.
Legal	8.5 x 14 inch
A4	210 x 295mm
B4	257 x 364mm
B5	18 x 20 cm
B	11 x 17 inch
Note	4 x 5inch

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

Part III - Object Classes

14 Introduction

This part describes all different object classes that are available in the Forms Library. All available object classes are summarized in a table below. 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_CLASS(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 upper 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_CLASS(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, there is the way in which the user interacts with the object, and second there's the question under which circumstances an object changes its state and how this is returned to the application program. All this is described in detail in the following sections.

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.

Object classes can be roughly divided into static object classes (Box, Frame, LabelFrame, Text, Bitmap, Pixmap, Clock and Chart), Buttons, valuator objects classes (Slider, Scrollbar, Dial, Positioner, Counter, Thumbwheel), Inputs, choice object classes (Menu, Choice, Browser), container object classes (Tabbed Folder, Form Browser, Menu bar) and, finally, other object classes (Timer, XYPlot, Canvas, Popup).

Box — Rectangular areas to visually group objects.

Frame — A box with an empty inside region.

LabelFrame — A frame with label on the frame.

Text — Simple one line labels.

Bitmap — Displays an X11 bitmap.

Pixmap — Displays a pixmap using the XPM library.

Clock — A clock.

Chart — Bar-charts, pie-charts, strip-charts, etc.

Button — Many different kinds and types of buttons that the user can push.

Slider
ValSlider — Both vertical and horizontal sliders to let the user indicate some float value, possibly with a field showing the currently set value.

Scrollbar — Sliders plus two directional buttons.

Dial — A dial to let the user indicate a float value.

Positioner — Lets the user indicate an (x, y) position with the mouse.

Counter — A different way to let a user step through values.

Thumbwheel
Rolling a wheel to indicate float values.

Input — Lets the user type in an input string.

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.

Folder — A (tabbed) folder is a compound object capable of holding multiple groups of objects.

FormBrowser
A browser you can drop forms into.

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 — Shows simple 2D xy-plot from a tabulated function or a datafile. Data points can be interactively manipulated and retrieved.

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.

Popups — Popup are mostly used by menus and choices, but they can also be used standalone to isplay context menus etc.

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 described what effect the routine `[fl_set_object_color()], page 287` has on the appearance of the object. Also some remarks on possible boxtypes are made.

15 Static Objects

15.1 Box Object

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.

15.1.1 Adding Box Objects

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 per default placed in the center of the box.

15.1.2 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 just 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.

15.1.3 Box Attributes

The first color argument (`col1`) to [`fl_set_object_color()`], page 287 controls the color of the box, the second (`col2`) is not used.

15.1.4 Remarks

No interaction takes place with boxes.

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

15.2 Frame Object

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

15.2.1 Adding Frame Objects

To add a frame to a form you use the routine

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

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

15.2.2 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.

15.2.3 Frame Attributes

The first color argument (`col1`) of `[fl_set_object_color()], page 287` controls the color of the frame if applicable, the second (`col2`) is not used. The boxtype attribute does not apply to the frame class.

15.2.4 Remarks

No interaction takes place with frames.

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

15.3 LabelFrame Object

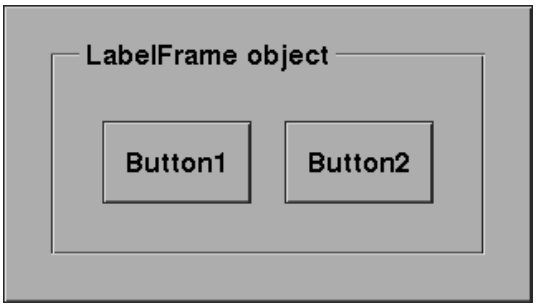
A label frame is almost the same as a frame except that the label is placed on the frame (See Fig. 15.1) instead of inside or outside of the bounding box as in a regular frame.

15.3.1 Adding LabelFrame Objects

To add a labelframe to a form you use the routine

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

The meaning of the parameters is as usual except that the frame is drawn outside of the bounding box (so a flat box of the same size just fills the inside of the frame without any gaps). The label is by default placed on the upper left hand part of the frame. Its position can changed (within limits) via calls of `[fl_set_object_lalign()], page 288`.



15.3.2 LabelFrame 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.

15.3.3 Attributes

The first color in the call of [`fl_set_object_color()`], page 287 controls the color of the frame if applicable. The second color controls the background color of the label. Boxtype attribute does not apply to the labelframe class

15.3.4 Remarks

No interaction takes place with labelframes.

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

15.4 Text Object

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

15.4.1 Adding Text Objects

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.

15.4.2 Text Types

Only one type of text exists: `FL_NORMAL_TEXT`.

15.4.3 Text Attributes

To set or change the text shown, use [`fl_set_object_label()`], page 288 or [`fl_set_object_label_f()`], page 288.

Any boxtype can be used for text.

The first color argument (`col1`) of [`fl_set_object_color()`], page 287 controls the color of the box the text is placed into, the second (`col2`) is not used. The color of the text itself is controlled by calls of [`fl_set_object_lcolor()`], page 288 as usual.

If the text is to change dynamically, boxtype `NO_BOX` should not be used for the object.

15.4.4 Remarks

No interaction takes place with text objects.

Don't use boxtype `FL_NO_BOX` if the text is to change dynamically. Note that there is almost no difference between a box with a label and a text. The only difference lies in the position where the text is placed and the fact that text is clipped to the bounding box. Text is normally placed inside the box at the left side. This helps you putting different lines of text below each other. Labels inside boxes are default centered in the box. You can change the position of the text inside the box using the routine `[fl_set_object_lalign()], page 288`. In contrast to boxes different alignments for text always place the text inside the box rather than outside the box.

15.5 Bitmap Object

A bitmap is a simple bitmap shown on a form.

15.5.1 Adding Bitmap Objects

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 is empty on creation.

15.5.2 Bitmap Types

Only the type `FL_NORMAL_BITMAP` is available.

15.5.3 Bitmap Interaction

No interaction takes place with a bitmap. For bitmaps that interact see Section 16.1 [Adding Button Objects], page 120, on how to create a button with a bitmap on top of it. (You can also place a hidden button over it if you want something to happen when pressing the mouse on a static bitmap.)

15.5.4 Other Bitmap Routines

To set the actual bitmap being displayed use

```
void fl_set_bitmap_data(FL_OBJECT *obj, int w, int h,
                        unsigned char *bits);
void fl_set_bitmap_file(FL_OBJECT *obj, const char *file);
```

`bits` contains the bitmap data as a character string. `file` is the name of the file that contains the 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 Bitmap 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 the other parameters have the obvious meanings. If there is no window created yet, the return value of `[fl_default_window()], page 254` may be used.

Note: bitmaps created by the above routines have a depth of 1 and should be displayed using `XCopyPlane()`.

15.5.5 Bitmap Attributes

The label color as set by `[fl_set_object_lcolor()], page 288` controls both the foreground color of the bitmap and the color of the label (i.e., they are always identical).

The first color argument (`col1`) to `[fl_set_object_color()], page 287` sets the background color of the bitmap (and the color of the box), the second (`col2`) is not used.

15.5.6 Remarks

See `demo33.c` for a demo of a bitmap.

15.6 Pixmap Object

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

15.6.1 Adding Pixmap Objects

To add a pixmap to a form 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 is empty on creation.

15.6.2 Pixmap Types

Only the type `FL_NORMAL_PIXMAP` is available.

15.6.3 Pixmap Interaction

No interaction takes place with a pixmap. For pixmap that interacts see Section 16.1 [Adding Button Objects], page 120, on how to create a button with a pixmap on top of it. (You can also place a hidden button over it if you want something to happen when pressing the mouse on a static pixmap.)

15.6.4 Other Pixmap Routines

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

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

```
void fl_set_pixmap_file(FL_OBJECT *obj, const char *file);
void fl_set_pixmap_data(FL_OBJECT *obj, char **data);
```

In the first routine, you specify the pixmap by the filename `file` that contains it. In the second routine, you `#include` the pixmap at compile time and use the pixmap data (an array of `char` pointers) directly. Note that both of these functions do not free the old

pixmaps associated with the object. If you're writing a pixmap browser type applications, be sure to free the old pixmaps by calling

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

on the pixmap object prior to calling these two routines. This function, in addition to freeing the pixmap and the mask, also frees the colors the pixmap allocated.

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

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

In some situations, you might already have a pixmap resource ID, e.g., from [fl_read_pixmapfile()], page 115 (see below in the "Remarks" subsection). Then you can use the following routine to change the the pixmap

```
void fl_set_pixmap_pixmap(FL_OBJECT *obj, Pixmap id,
                          Pixmap mask);
```

where mask is used for transparency (see [fl_read_pixmapfile()], page 115). 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 function [fl_free_pixmap_pixmap()], page 115.

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 *obj, int align,
                         int dx, int dy);
```

where align is the same as that used for labels, see Section 3.11.3 [Label Attributes and Fonts], page 27, 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.

15.6.5 Pixmap 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 the color found being smaller than some preset threshold values between 0 and 65535 (0 means exact match). The default thresholds are 40000 for red, 30000 for green and 50000 for blue. To change these defaults, use the following routine

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

15.6.6 Remarks

The following routines may be handy for reading 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, returned by a call of [fl_default_window()],

page 254. Parameter shape_mask is a pointer to an already existing Pixmap, which, if not NULL, is used as a clipping mask to achieve transparency. hotx and hoty are the center of the pixmap (useful if the pixmap is to be used as a cursor). Parameter tran is currently not used.

If you already have the pixmap data in memory, the following routine can 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 for fl_read_pixmapfile.

Note that the Forms Library handles transparency, if specified in the pixmap file or data, for pixmap and pixmapbutton objects. However, when using [fl_read_pixmapfile()], page 115 or [fl_create_from_pixmapdata()], page 116, the application programmer is responsible to set the clip mask in an appropriate GC.

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

```
void fl_free_pixmap(Pixmap id);
```

You will need the XPM library (version 3.4c or later)m developed by Arnaud Le Hors and Groupe Bull, to use pixmaps. The XPM library is avalialble as a package for most distributions, but can also be obtained from many X mirror sites, e.g., via anonymous FTP from (ftp://ftp.x.org/contrib/libraries/. Its home page is http://old.koalateam.com/lehors/xpm.html.

15.7 Clock Object

A clock object simply displays a clock on the form

15.7.1 Adding Clock Objects

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 placed below the clock by default.

15.7.2 Clock Types

The following types are available:

FL_ANALOG_CLOCK
: An analog clock complete with the second hand.

FL_DIGITAL_CLOCK
: A digital clock.

15.7.3 Clock Interaction

No interaction takes place with clocks.

15.7.4 Other Clock Routines

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

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

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

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

```
long fl_set_clock_adjustment(FL_OBJECT *obj, long adj);
```

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

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

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

15.7.5 Clock Attributes

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

The first color argument (`col1`) of `[fl_set_object_color()], page 287` controls the color of the background, the second (`col2`) is the color of the hands.

15.7.6 Remarks

See `flclock.c` for an example of the use of clocks. See Section 33.1 [Misc. Functions], page 302, for other time related routines.

15.8 Chart Object

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 processes.

15.8.1 Adding Chart Objects

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.

15.8.2 Chart 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 the 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 less then or equal to 0. The maximum number of values displayed in the chart can be set using the routine [fl_set_chart_maxnumb()], page 119. The argument must be not larger than `FL_CHART_MAX` which currently is 512. Switching between different types can be done without any complications.

15.8.3 Chart Interaction

No interaction takes place with charts.

15.8.4 Other Chart 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, FL_COLOR 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 into the colormap (`FL_RED` etc.) that is the color of this item. The chart will be redrawn each time you add an item. This might not be appropriate if you are filling a chart with values. In this case put the calls between calls of [fl_freeze_form()], page 289 and [fl_unfreeze_form()], page 289.

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

```
void fl_set_chart_lstyle(FL_OBJECT *obj, int fontstyle);
void fl_set_chart_lsize(FL_OBJECT *obj, int fontsize);
void fl_set_chart_lcolor(FL_OBJECT *obj, FL_COLOR color);
```

Note that [fl_set_chart_lcolor()], page 118 only affects the label color of subsequent items, not the items already created.

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

```
void fl_insert_chart_value(FL_OBJECT *obj, int index,
                           double val, const char *text,
                           FL_COLOR 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,
                            FL_COLOR 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 call it with both `min` and `max` being set to `0.0`. To obtain the current bounds, use the following routine

```
void fl_get_chart_bounds(FL_OBJECT *obj, double *min, double *max)'
```

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` being set to false (0). In this case the width of the bars will be such that the maximum number of items fits in the box. This maximal number (defaults to `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, which may not be larger than `FL_CHART_MAX`.

15.8.5 Chart Attributes

Don't use boxtype `FL_NO_BOX` for a chart object if it changes value.

Normally, for bar and line chart a baseline is drawn at 0. This can be switched on and off by the function

```
void fl_set_chart_baseline(FL_OBJECT *ob, int yes_no);
```

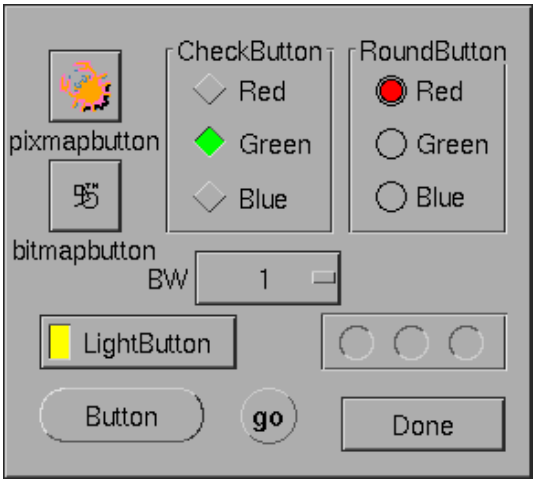
The first color argument (`col1`) to `[fl_set_object_color()], page 287` controls the (background) color of the box, the second (`col2`) the color of the baseline.

15.8.6 Remarks

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

16 Button-like Objects

A very important set of object classes are those for buttons. Buttons are placed on the form such that the user can push them with the mouse. The different button classes mostly are distinguished by the way they are displayed. Differences in behaviour can be achieved by using different types for a button: there exist button types that make them return to their normal state when the user releases the mouse, types for buttons that stay pushed until the user pushes them again, a radio button type for buttons that are grouped with other radio buttons and of which only one can be in the on state at a time and a touch button type for buttons that "fire" repeatedly while being pressed.



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 these buttons switches the light on. Round buttons are simple circles and, when pushed, a colored circle appears inside of them. Bitmap and pixmap buttons are buttons with an image in addition to a text label.

16.1 Adding Button Objects

Adding an object To add buttons use one of the following routines:

```
FL_OBJECT *fl_add_button(int type, FL_Coord x, FL_Coord y,
                         FL_Coord w, FL_Coord h,
                         const char *label);
FL_OBJECT *fl_add_lightbutton(int type, FL_Coord x, FL_Coord y,
                              FL_Coord w, FL_Coord h,
                              const char *label);
FL_OBJECT *fl_add_roundbutton(int type, FL_Coord x, FL_Coord y,
                              FL_Coord w, FL_Coord h,
                              const char *label);
FL_OBJECT *fl_add_round3dbutton(int type, FL_Coord x, FL_Coord y,
                                FL_Coord w, FL_Coord h,
                                const char *label);
FL_OBJECT *fl_add_checkbutton(int type, FL_Coord x, FL_Coord y,
```

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

These finctions create buttons of the following classes:

`FL_BUTTON`
: A standard normal button.

`FL_LIGHTBUTTON`
: A button with a small embedded, colored area that changes color when the button is in the on state.

`FL_ROUNDBUTTON`
: A circular button (with a label beside). The inner area of the circle changes color when the button is on. Often used for radio buttons.

`FL_ROUND3DBUTTON`
: Just like the `[FL_ROUNDBUTTON], page 121` but the circle is dran in a 3D-fashion.

`FL_CHECKBUTTON`
: Button shaped in the form of a rhombus slightly raised above the forms plane when off and slightly embossed (ypically with a different color) when on.

`FL_BITMAPBUTTON`
: Button decorated with a bitmap (often read in from an X bitmap file with extension `xbm`) in additional to a label.

`FL_PIXMAPBUTTON`
: Button decorated with a pixmap (often read in from an X pixmap file with extension `xpm`) in additional to a label. An additonal pixmap can be set for the case that the mouse hoovers over the button.

`FL_LABELBUTTON`
: A button that does not appear to be a button, only its label is shown, can be used e.g., for hyperlinks.

`FL_SCROLLBARBUTTON`
: A button mostly used at the ends of scrollbars - instead of a label it can only contain a triangle pointing up, down, left or right.

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

16.2 Button Types

The following types of buttons are available:

`FL_NORMAL_BUTTON`
: Returned to [`fl_do_forms()`], page 296 (or having its callback funtion invoked) when released.

`FL_PUSH_BUTTON`
: Stays pushed until user pushes it again.

`FL_MENU_BUTTON`
: Returned when pushed, useful e.g., for opening a popup when pushed.

`FL_TOUCH_BUTTON`
: Returned at regulat intervals 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`
: Returned both when pushed and when released.

`FL_RETURN_BUTTON`
: Like a normal button but also reacts to the `<Return>` key.

`FL_HIDDEN_RET_BUTTON`
: Invisible return button.

Except for the [`FL_HIDDEN_BUTTON`], page 122 and [`FL_HIDDEN_RET_BUTTON`], page 122, which are invisible, there's not much visible indication of the button type but the function is quite different. For each of the types the button gets pushed down when the user presses the mouse on top of it. What actually happens when the user does so then depends on the type of the button. An [`FL_NORMAL_BUTTON`], page 122, [`FL_TOUCH_BUTTON`], page 122 and [`FL_INOUT_BUTTON`], page 122 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`], page 122 remains pushed and is only released when the user pushes it again. A [`FL_RADIO_BUTTON`], page 122 is a push button with the following extra property: whenever the user pushes a radio button, all other pushed radio buttons in the same form (or in the same group) are released. In this way the user can make its choice among several possibilities. A [`FL_RETURN_BUTTON`], page 122 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], page 122 behaves like a normal button but is invisible. A [FL_HIDDEN_RET_BUTTON], page 122 is like a hidden button but also reacts to <Return> key presses.

16.3 Button Interaction

[FL_NORMAL_BUTTON], page 122s, [FL_PUSH_BUTTON], page 122s, [FL_RADIO_BUTTON], page 122s, [FL_RETURN_BUTTON], page 122s and [FL_HIDDEN_BUTTON], page 122s are returned at the moment the user releases the mouse after having pressed it on the button. A [FL_MENU_BUTTON], page 122, in contrast, is returned already on a mouse press. A [FL_INOUT_BUTTON], page 122 is returned both when the user presses it and when the user releases it. A [FL_TOUCH_BUTTON], page 122 is returned all the time as long as the user keeps the mouse button pressed while the mouse is on top of it. A [FL_RETURN_BUTTON], page 122 and a [FL_HIDDEN_RET_BUTTON], page 122 are also returned when the user presses the <Return> key.

As for other "active" objects, you can control under which conditions a button object gets returned or its callback invoked by using the function

```
int fl_set_object_return(FL_OBJECT *obj, unsigned int when);
```

where reasonable values for `when` are

[FL_RETURN_NONE], page 45
: Never return object or invoke callback.

[FL_RETURN_END_CHANGED], page 45
: Return object or invoke callback when mouse button is released and at the same moment the state of the button changed.

[FL_RETURN_CHANGED], page 44
: Return object or invoke callback whenever the state of the button changes.

[FL_RETURN_END], page 44
: Return object or invoke callback when mouse button is released

[FL_RETURN_ALWAYS], page 45
: Return object or invoke callback on all of the above condtions.

Most buttons will always return [FL_RETURN_END], page 44 and [FL_RETURN_CHANGED], page 44 at the same time. Exceptions are [FL_INOUT_BUTTON], page 122s and [FL_TOUCH_BUTTON], page 122s. The former returns [FL_RETURN_CHANGED], page 44 when pushed and both [FL_RETURN_END], page 44 and [FL_RETURN_CHANGED], page 44 together when released. [FL_TOUCH_BUTTON], page 122s return when pressed, then [FL_RETURN_CHANGED], page 44 at regular time intervals while being pressed and finally [FL_RETURN_END], page 44 when released.

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

16.4 Other Button 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 set to be pushed (1) or released (0). When setting a [FL_RADIO_BUTTON], page 122 to be pushed this automatically releases the currently pushed radio button in the same form (or group). Also note that this routine only simulates the visual appearance but does not affect the program flow in any way, i.e., setting a button as being pushed does not invoke its callback or results in the button becoming returned to the program. For that follow up the call of [fl_set_button()], page 124 with a call of [fl_trigger_object()], page 290 (or [fl_call_object_callback()], page 290).

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

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

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

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

It returns one of the constants [FL_LEFT_MOUSE], page 243, [FL_MIDDLE_MOUSE], page 243, [FL_RIGHT_MOUSE], page 243, [FL_SCROLLUP_MOUSE], page 243 or [FL_SCROLLDOWN_MOUSE], page 243 (the latter two are from the scroll wheel of the mouse). If the last push was triggered by a shortcut (see below), the function returns the `keysym` (ASCII value if the key used is between 0 and 127) of the key plus [FL_SHORTCUT], page 244. For example, if a button has `<Ctrl>-C` as its shortcut the button number returned upon activation of the shortcut will be `FL_SHORTCUT + 3` (the ASCII value of `<Ctrl>-C` is 3).

It can also be controlled which mouse buttons a buttons reacts to (per default a button reacts to all mouse buttons, including the scroll wheel). To set which mouse buttons the button reacts to use

```
void fl_set_button_mouse_buttons(FL_OBJECT *obj, int mbuttons);
```

`mbuttons` is the bitwise OR of the numbers 1 for the left mouse button, 2 for the middle, 4 for the right mouse button, 8 for moving the scroll wheel up "button" and 16 for scrolling down "button". Per default a button reacts to all mouse buttons.

To determine which mouse buttons a button is reacting to use

```
void fl_get_button_mouse_buttons(FL_OBJECT *obj,
                                 unsigned int *mbuttons);
```

The value returned via `mbuttons` is the same value as would be used in [fl_set_button_mouse_buttons()], page 124.

In a number of situations it is useful to define a keyboard equivalent for a button. You might e.g., want to define that `<Ctrl>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,
```

[1] [fl_mouse_button()], page 50 can also be used.

```
int showUL);
```

Note that `str` is a string, not a single character. This string is a list of all the characters to become keyboard shortcuts for the button. E.g., if you use string `"^QQq"` the button will react on the keys `q`, `Q` and `<Ctrl>Q`. (As you see you can 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 are always evaluated before input fields. Other special keys, such as `<F1>` etc., can also be used as shortcuts. See Section 26.1 [Shortcuts], page 245, for details. Finally, keep in mind that a button of type `FL_RETURN_BUTTON` is in fact nothing more than a normal button, just with the `<Return>` key set as the shortcut. So don't change the shortcuts for such a button.

If the third 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 the letter `a` or `A` it will be underlined (i.e., special characters such as `#` and `^` are ignored when matching). A false value (0) for `showUL` turns off underlining without affecting the shortcut. Note that although the entire object label is searched for matching character to underline of the shortcut string itself only the first (non-special) character is considered, thus a shortcut string of `"Yy"` for the label `"Yes"` will result in the letter `Y` becoming underlined while for `"yY"` it won't.

To set the bitmap to use for a bitmap button the following functions can be used:

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

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

```
void fl_set_pixmapbutton_data(FL_OBJECT *obj, unsigned char **bits);
void fl_set_pixmapbutton_file(FL_OBJECT *obj, const char *file);
void fl_set_pixmapbutton_pixmap(FL_OBJECT *obj, Pixmap id,
                                Pixmap mask);
```

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

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

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

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

```
Pixmap fl_get_pixmapbutton_pixmap(FL_OBJECT *obj,
                                  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 *obj, int align,
```

```
                                        int xmargin, int ymargin);
```

where `align` is the same as that used for labels. See Section 3.11.3 [Label Attributes and Fonts], page 27, for a list. `xmargin` and `ymargin` are extra margins to leave in addition to the object border width. Note that although you can place a pixmap outside of the bounding box, it probably is not a good idea.

When the mouse enters a pixmap button an outline of the button is shown. If required, a different pixmap (the focus pixmap) can also be shown. To set such a focus pixmap the following functions are available:

```
void fl_set_pixmapbutton_focus_data(FL_OBJECT *obj,
                                    unsigned char **bits);
void fl_set_pixmapbutton_focus_file(FL_OBJECT *obj,
                                   const char *file);
void fl_set_pixmapbutton_focus_pixmap(FL_OBJECT *obj, Pixmap id,
                                      Pixmap mask);
```

The meanings of the parameters are the same as that in the regular pixmap routines.

Finally, there's a function that can be used to enable or disable the focus outline

```
void fl_set_pixmapbutton_focus_outline(FL_OBJECT *obj, int yes_no);
```

See also Section 15.6 [Pixmap Object], page 114, for pixmap color and transparency handling.

To get rid of a focus pixmap of a pixmap button use the function

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

16.5 Button Attributes

For normal buttons the first color argument (`col1`) to `[fl_set_object_color()], page 287` controls the normal color and the second (`col2`) the color the button has when pushed. For lightbuttons `col1` is the color of the light when off and `col2` the color when on. For round buttons, `col1` is the color of the circle and `col2` the color of the circle that is placed inside it when pushed. For round3dbutton, `col1` is the color of the inside of the circle and `col2` the color of the embedded circle. For bitmapbuttons, `col1` is the normal box color (or bitmap background if boxtype is not `FL_NO_BOX`) and `col2` is used to indicate the focus color. The foreground color of the bitmap is controlled by label color (as set via `[fl_set_object_lcolor()], page 288`. For scrollbutton, `col1` is the overall boundbox color (if boxtype is not `FL_NO_BOX`), `col2` is the arrow color. The label of a scrollbutton must be a string with a number between 1 and 9 (except 5), indicating the arrow direction like on the numerical key pad. The label can have an optional prefix `#` to indicate uniform scaling. For example, the label `"#9"` tells that the arrow should be pointing up-right and the arrow has the identical width and height regardless the overall bounding box size.

16.6 Remarks

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

17 Valuator Objects

17.1 Slider Object

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

17.1.1 Adding Slider Objects

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 difference between a normal slider and a valslider is that for the second type its value is displayed above or to the left of the slider.

17.1.2 Slider Types

The following types of sliders are available:

`FL_VERT_SLIDER`
: A vertical slider.

`FL_HOR_SLIDER`
: A horizontal slider.

`FL_VERT_FILL_SLIDER`
: A vertical slider, filled from the bottom.

`FL_HOR_FILL_SLIDER`
: A horizontal slider, filled from the left.

`FL_VERT_NICE_SLIDER`
: A nice looking vertical slider.

`FL_HOR_NICE_SLIDER`
: A nice looking horizontal slider.

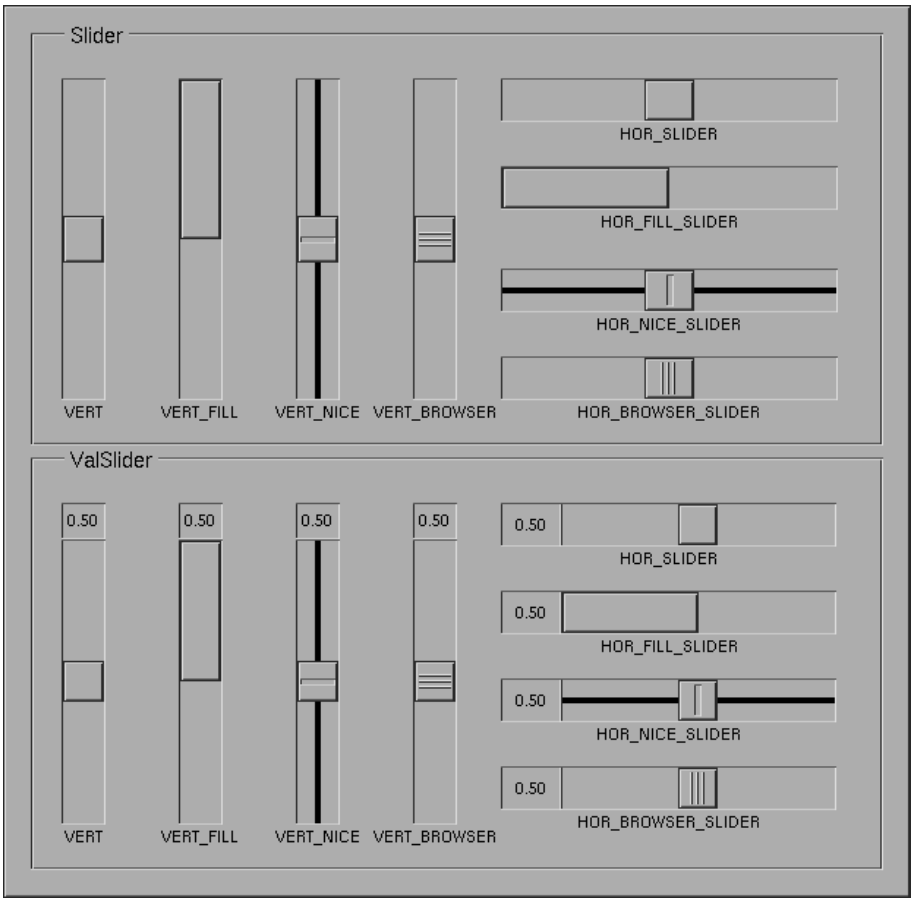
`FL_VERT_BROWSER_SLIDER`
: A different looking vertical slider.

`FL_HOR_BROWSER_SLIDER`
: A different looking horizontal slider.

`FL_VERT_PROGRESS_BAR`
: A vertical progress bar

`FL_HOR_PROGRESS_BAR`
A horizontal progress bar

Please note that except for `[FL_VERT_PROGRESS_BAR], page 127` and `[FL_HOR_PROGRESS_BAR], page 127` the label will always drawn on the outside of the slider (even if you attempt to set an inside alignment).



17.1.3 Slider Interaction

Whenever the user changes the value of the slider using the mouse, the slider is returned (unless there's callback function associated with the object) by the interaction routines. The slider position is changed by moving the mouse inside the slider area. For fine control, hold down the `<Shift>` key while moving the slider.

Please note: the `[FL_VERT_PROGRESS_BAR], page 127` and `[FL_HOR_PROGRESS_BAR], page 127` aren't actually valuator objects (they don't react to any user interaction) but are vor visualization only (i.e., showing a progress bar that is changed by the program only), they appear hear because they are directly derived from the `[FL_VERT_FILL_SLIDER], page 127` and `[FL_VERT_FILL_SLIDER], page 127` slider. Thus the only way to change the value of objects of these types is by calling `[fl_set_slider_value()], page 129`! To obtain the correct "progress bar" behaviour you should also update the label accordingly.

In some cases you might not want the slider to be returned or its callback called each time its value changes. To change the default, call the following routine:

```
void fl_set_object_return(FL_OBJECT *obj, unsigned int when)
```

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

`[FL_RETURN_NONE], page 45`
: Never return or invoke callback.

`[FL_RETURN_END_CHANGED], page 45`
: Return or invoke callback at end (mouse release) if value is changed since last return.

`[FL_RETURN_CHANGED], page 44`
: Return or invoke callback whenever the slider value is changed. This is the default.

`[FL_RETURN_END], page 44`
: Return or invoke callback at end (mouse release) regardless if the value is changed or not.

`[FL_RETURN_ALWAYS], page 45`
: Return or invoke callback when the value changed or at end (mouse release).

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

17.1.4 Other Slider Routines

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

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

By default, the minimum value for a slider is 0.0, the maximum is 1.0 and the value is 0.5. For vertical sliders the slider position for the minimum value is at the left, for horizontal sliders at the top of the slider. By setting `nin` to a larger value than `max` in a call of `[fl_set_slider_bounds()], page 129` this can be reversed.

If in a call of `[fl_set_slider_bounds()], page 129` the actual value of a slider isn't within the range of the new bounds, its value gets adjusted to the nearest limit. When the requested new slider value in a call of `[fl_set_slider_value()], page 129` is outside the range of bounds it gets adjusted to the nearest boundary value.

To obtain the current value or bounds of a slider use

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

17.1.5 Slider Attributes

Never use `FL_NO_BOX` as the boxtype for a slider. For `FL_VERT_NICE_SLIDER`s and `FL_HOR_NICE_SLIDER`s it's best to use a `FL_FLAT_BOX` in the color of the background to get the nicest effect.

The first color argument (`col1`) to `[fl_set_object_color()], page 287` controls the color of the background of the slider, the second (`col2`) 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 floating point value between 0.0 and 1.0. The default is `FL_SLIDER_WIDTH`, which is 0.1 for regular sliders and 0.15 for browser sliders. With a value for `size` of 1.0, the slider covers the box completely and can no longer be moved. This function does nothing if applied to sliders of type `NICE_SLIDER` and `FILL_SLIDER`.

To obtain the current setting of the slider size use

```
double fl_get_slider_size(FL_OBJECT *obj);
```

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, i.e., valsliders. The argument must be between 0 and `FL_SLIDER_MAX_PREC` (currently set to 10).

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

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

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

```
const char *filter(FL_OBJECT *obj, double value, int prec) {
    static char buf[32];

     sprintf(buf, "%.*f", prec, value);
     return buf;
}
```

Finally, the function

```
int fl_get_slider_repeat(FL_OBJECT *obj);
void fl_set_slider_repeat(FL_OBJECT *obj, int millisec)
```

allows to determine and control the time delay (in milliseconds) between jumps of the sliders knob when the mouse button is kept pressed down on the slider outside of the knobs area. The default value is 100 ms. The delay for the very first jump is twice that long in order to avoid jumping to start too soon when only a single click was intended but the user is a bit slow in releasing the mouse button.

17.1.6 Remarks

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

17.2 Scrollbar Object

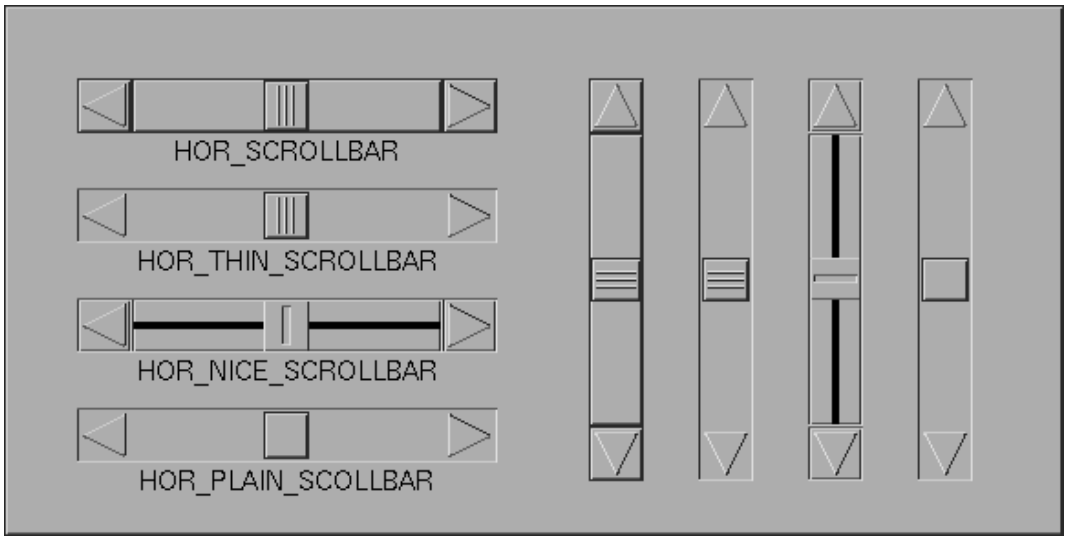
Scrollbars are similar to sliders (as a matter of fact, scrollbars are made with sliders and scrollbuttons), and useful in letting the user indicate a value between some fixed bounds. Both horizontal and vertical scrollbars exist. They have a minimum, maximum and current value (all floating point values).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

17.2.1 Adding Scrollbar Objects

To add a scrollbar to a form use

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

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



17.2.2 Scrollbar Types

The following types of scrollbar are available:

`FL_VERT_SCROLLBAR`
: A vertical scrollbar.

`FL_HOR_SCROLLBAR`
: A horizontal scrollbar.

`FL_VERT_THIN_SCROLLBAR`
: A different looking vertical scrollbar.

`FL_HOR_THIN_SCROLLBAR`
: A different looking horizontal scrollbar.

`FL_VERT_NICE_SCROLLBAR`
: A vertical scrollbar using `FL_NICE_SLIDER`.

`FL_HOR_NICE_SCROLLBAR`
: A horizontal scrollbar using `FL_NICE_SLIDER`.

`FL_VERT_PLAIN_SCROLLBAR`
: Similar to `FL_THIN_SCROLLBAR`.

`FL_HOR_PLAIN_SCROLLBAR`
: Similar to `FL_HOR_THIN_SCROLLBAR`.

17.2.3 Scrollbar Interaction

Whenever the user changes the value of the scrollbar, the scrollbar's callback is called (if one is associated with the scrollbar). The scrollbar position can be changed in several ways. The most simple one is to left-click on the knob of the scrollbar and move the know while the left mouse button is kept pressed down. Left-clicking beside the know will move the knob in large steps toward the current position of the mouse, clicking with the middle or right mouse button in smaller steps. Small shifts can also be obtained by clicking on one of the buttons at the side of the scrollbar or by using the scroll-wheel somehwere over the scrollbar.

You can control under which conditions the scrollbar gets returned to your application or its callback invoked. To change the default, call

```
void fl_set_object_return(FL_OBJECT *obj, unsigned int when);
```

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

`[FL_RETURN_NONE], page 45`
: Never return or invoke callback.

`[FL_RETURN_END_CHANGED], page 45`
: Return or invoke callback at end (mouse release) if value is changed (since last return).

`[FL_RETURN_CHANGED], page 44`
: Return or invoke callback whenever the scrollbar value is changed. This is the default.

`[FL_RETURN_END], page 44`
: Return or invoke callback at end (mouse release) regardless if the value is changed or not.

`[FL_RETURN_ALWAYS], page 45`
: Return or invoke callback whenever value changed or mouse button was released.

The default setting for `when` for a scrollbar object is `[FL_RETURN_CHANGED], page 44` (unless during the build of XForms you set the configuration flag `--enable-bwc-bs-hack` in which case the default is `[FL_RETURN_NONE], page 45` to keep backward compatibility with earlier releases of the library).

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

17.2.4 Other Scrollbar Routines

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

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

By default, the minimum value for a slider is 0.0, the maximum is 1.0 and the value is 0.5. For vertical sliders the slider position for the minimum value is at the left, for horizontal sliders at the top of the slider. By setting `min` to a larger value than `max` in a call of `[fl_set_scrollbar_bounds()], page 133` this can be reversed.

If in a call of `[fl_set_scrollbar_bounds()], page 133` the actual value of a scrollbar isn't within the range of the new bounds, its value gets adjusted to the nearest limit. When the requested new scrollbar value in a call of `[fl_set_scrollbar_value()], page 133` is outside the range of bounds it gets adjusted to the nearest boundary value.

To obtain the current value and bounds of a scrollbar use

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

By default, if the mouse is pressed beside the the sliding bar, the bar starts to jumps in the direction of the mouse position. You can use the following routine to change this size of the steps being made :

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

where `lj` indicates how much to increment if the left mouse button is pressed and `rj` indicates how much to jump if the middle mouse button pressed. For example, for the scrollbar in the browser class, the left mouse jump is made to be one page and middle mouse jump is made to be one line. The increment (decrement) value when the scrollbuttons are pressed is set to the value of the right jump. The default values for `lj` and `rj` are `0.1` and `0.02`.

To obtain the current increment settings, use the following routine

```
void fl_get_scrollbar_increment(FL_OBJECT *obj, double *lj, double *sj);
```

17.2.5 Scrollbar Attributes

Never use `FL_NO_BOX` as the boxtype for a scrollbar. For `FL_VERT_NICE_SCROLLBAR`s and `FL_HOR_NICE_SCROLLBAR`s it's best to use a `FL_FLAT_BOX` boxtype in the color of the background to get the nicest effect.

The first color argument (`col1` to `[fl_set_object_color()], page 287` controls the color of the background of the scrollbar, the second (`col2`) the color of the sliding bar itself.

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

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

`size` should be a value between 0.0 and 1.0. The default is `FL_SLIDER_WIDTH`, which is 0.15 for all scrollbars With `size` set to 1.0, the scrollbar covers the box completely and can no longer be moved. This function does nothing if applied to scrollbars of type `FL_NICE_SCROLLBAR`.

The function

```
double fl_get_scrollbar_size(FL_OBJECT *obj);
```

returns the current setting of the scrollbar size.

17.2.6 Remarks

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

17.3 Dial Object

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 floating point values). The user can change this value by turning the dial with the mouse. Whenever the value changes, this is reported to the application program.

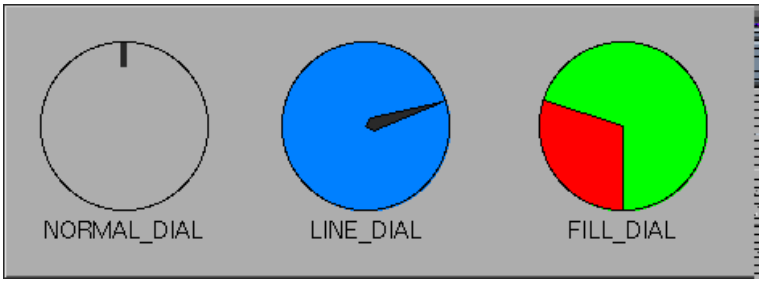
17.3.1 Adding Dial Objects

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.

17.3.2 Dial Types



The following types of dials are available:

`FL_NORMAL_DIAL`
: A dial with a knob indicating the position.

`FL_LINE_DIAL`
: A dial with a line indicating the position.

`FL_FILL_DIAL`
: The area between initial and current is filled.

17.3.3 Dial Interaction

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

```
void fl_set_object_return(FL_OBJECT *obj, unsigned int when);
```

where `when` can be one of the following

`[FL_RETURN_NONE], page 45`
: Never report or invoke callback.

`[FL_RETURN_END_CHANGED], page 45`
: Return or invoke callback at end (mouse release) and only if the dial value is changed. This is the default setting.

[FL_RETURN_CHANGED], page 44
: Return or invoke callback whenever the dial value is changed.

[FL_RETURN_END], page 44
: Return or invoke callback at the end regardless if the dial value is changed or not.

[FL_RETURN_ALWAYS], page 45
: Return or invoke callback when value has changed or mouse button has been released.

17.3.4 Other Dial Routines

To change the value of the dial and its bounds 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 and its bounds use

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

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

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

where `thetai` maps to the minimum value of the dial and `thetaf` to its maximum value. The angles are relative to the origin of the dial, which is by default at 6 o'clock and rotates clock-wise. By default, the minimum angle is 0 and the maximum angle is 360.

To obtain the start and end angles use

```
void fl_get_dial_angles(FL_OBJECT *obj, double *thetai, double *thetaf)
```

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

```
void fl_set_dial_crossover(FL_OBJECT *obj, int yes_no);
```

where a true value for `yes_no` indicates that cross-over is allowed.

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 a value of 0.0 for `step` to switch off rounding.

To get the current setting for the rounding steps use

```
double fl_get_dial_step(FL_OBJECT *obj);
```

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

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

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

To obtain the direction use

```
int fl_get_dial_direction(FL_OBJECT *obj);
```

17.3.5 Dial Attributes

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

The first color argument (`col1` to `[fl_set_object_color()], page 287` controls the color of the background of the dial, the second `col2`) the color of the knob or the line or the fill color.

17.3.6 Remarks

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

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

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

17.4 Positioner Object

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 (except an invisble positioner, of course). Clicking the mouse inside the box changes the position of the cross-hair cursor and, hence, the x- and y-values.

17.4.1 Adding Positioner Objects

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.

17.4.2 Positioner Types

The following types of positioner exist:

`FL_NORMAL_POSITIONER`
: Cross-hair inside a box.

`FL_OVERLAY_POSITIONER`
: Cross-hair inside a transparent box (i.e.,drawn in in XOR mode).

`FL_INVISIBLE_POSITIONER`
: Completely invisible positioner, to be used just for the side effect of obtaining a position (typically an object is below below it that otherwise would receive user events).

17.4.3 Positioner Interaction

The user changes the setting of the positioner using the mouse inside the box. Per default whenever the values changes, the object is returned by the interaction routines or its callback invoked (if one exists.

To change the default use the function

```
void fl_set_object_return(FL_OBJECT *obj, unsigned int when);
```

where `when` can be one of the following

`[FL_RETURN_NONE], page 45`
: Never report or invoke callback.

`[FL_RETURN_END_CHANGED], page 45`
: Return or invoke callback at end (mouse release) and only when the positioner ended in a different position than the one it started from.

`[FL_RETURN_CHANGED], page 44`
: Return or invoke callback whenever the positioners value is changed, default setting.

`[FL_RETURN_END], page 44`
: Return or invoke callback at the end only but regardless if the positioners value changed or not.

`[FL_RETURN_ALWAYS], page 45`
: Return or invoke callback when value has changed or mouse button has been released.

Per default a positioner only reacts to the left mouse button. But sometimes it can be useful to modify which mouse buttons it will reacts to. To set this use

```
void fl_set_positioner_mouse_buttons(FL_OBJECT *obj,
                                     int mbuttons);
```

`mbuttons` is the bitwise OR of the numbers 1 for the left mouse button, 2 for the middle, 4 for the right mouse button, 8 for moving the scroll wheel up "button" and 16 for scrolling down "button". Per default a button reacts to all mouse buttons.

To determine which mouse buttons a positioner reacts to use

```
void fl_get_positioner_mouse_buttons(FL_OBJECT *obj,
                                     unsigned int *mbuttons);
```

The value returned via `mbuttons` is the same value as would be used in `[fl_set_positioner_mouse_buttons()], page 137`.

Sometimes you may want to assign different meanings to the mouse buttons used to interact with the positioner. To find out which one has been used there's the function

```
int fl_get_positioner_numb(FL_OBJECT *obj);
```

It returns one of the constants `[FL_LEFT_MOUSE], page 243`, `[FL_MIDDLE_MOUSE], page 243`, `[FL_RIGHT_MOUSE], page 243`, `[FL_SCROLLUP_MOUSE], page 243` or `[FL_SCROLLDOWN_MOUSE], page 243` (the latter two are from the scroll wheel of the mouse).

17.4.4 Other Positioner 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 actual values are 0.5. For boundaries in x-direction `min` and `max` should be taken to mean the left- and right-most position, respectively, and for the y-boundaries `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 and the bounds 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 might 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 a value of 0.0 for `step` to switch off rounding.

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

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

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

17.4.5 Positioner Attributes

Never use `FL_NO_BOX` as the boxtype for a positioner of type. `FL_NORMAL_POSITIONER` (but the other two types will have a box type of `FL_NO_BOX` per default).

The first color argument (`col1`) to `[fl_set_object_color()], page 287` controls the color of the box, the second (`col2`) the color of the cross-hair.

17.4.6 Remarks

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

17.5 Counter Object

A counter provides a different mechanism for the user to select a value. In consists of a box displaying a value with one or two buttons on each side. The user can press these buttons to change the value (and while the mouse button is kept pressed down the value will continue to change, slow at first and faster after some time). If the counter has four buttons, the

left- and right-most button make the value change in large steps, the other buttons make it change in small steps.

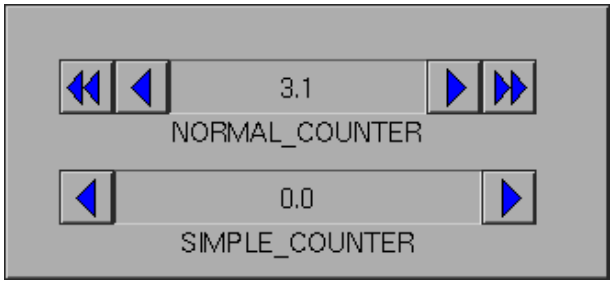
17.5.1 Adding Counter Objects

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.

17.5.2 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.

17.5.3 Counter Interaction

The user changes the value of the counter by keeping his mouse pressed on one of the buttons. Per default whenever the mouse is released and the counter value is changed the counter is returned to the application program or its callback is invoked.

In some applications you might want the counter to be returned to the application program (or the callback invoked) e.g., whenever the value changes and not only when the mouse is released. To this end use

```
void fl_set_object_return(FL_OBJECT *obj, unsigned int when);
```

where `when` can be either

`[FL_RETURN_NONE], page 45`
: Never report or invoke callback.

`[FL_RETURN_END_CHANGED], page 45`
: Return or invoke callback at end (mouse release) and only if the counter value is changed.

`[FL_RETURN_CHANGED], page 44`
: Return or invoke callback whenever the counter value is changed. This is the default setting.

[FL_RETURN_END], page 44
: Return or invoke callback at the end regardless if the counter value is changed or not.

[FL_RETURN_ALWAYS], page 45
: Return or invoke callback when the counter value has changed or mouse button has been released.

17.5.4 Other Counter Routines

To change the value of the counter, it's bounds and stp size use the routines

```
void fl_set_counter_value(FL_OBJECT *obj, double val);
void fl_set_counter_bounds(FL_OBJECT *obj, double min, double max);
void fl_set_counter_step(FL_OBJECT *obj, double small, double large);
```

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

For conflicting settings, bounds take precedence over value, i.e., if setting a value that is outside of the current bounds, it is clamped. Also changing the bounds in a way that the current counter value isn't within the new bounds range anymore will result in its value being adjusted to the nearest of the new limits.

To obtain the current value of the counter use

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

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

```
void fl_get_counter_bounds(FL_OBJECT *obj, double *min, double *max);
void fl_get_counter_step(FL_OBJECT *obj, double *small, double *large);
```

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

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

To determine the current value of the precision use

```
int fl_get_counter_precision(FL_OBJECT *obj);
```

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 *obj, double value, int prec) {
    static char buf[32];

     sprintf(buf, "%.*f",prec,value);
```

```
        return buf;
    }
```

By default the counter value changes first slowly and the rate of change then accelerates until a final speed is reached. The default delay between the value changing is 600 ms at the start and the final delay is 50 ms. You can change the initial delay by a call of the function

```
void fl_set_counter_repeat(FL_OBJECT *obj, int millisec);
```

and the final delay by using

```
void fl_set_counter_min_repeat(FL_OBJECT *obj, int millisec);
```

where in both cases the argument `millisec` is the delay in milli-seconds. The current settings for the initial and final delay can be obtained by calling the functions

```
int fl_get_counter_repeat(FL_OBJECT *obj);
int fl_get_counter_min_repeat(FL_OBJECT *obj);
```

Until version 1.0.91 of the library the delay between changes of a counter was constant (with a default value of 100 ms). To obtain this traditional behaviour simple set the initial and final delay to the same value.

As a third alternative you can also request that only the first change of the counter has a different delay from all the following ones. To achieve this call

```
void fl_set_counter_speedjump(FL_OBJECT *obj, int yes_no);
```

with a true value for `yes_no`. The delay for the first change of the counter value will then be the one set by `[fl_set_counter_repeat()], page 141` and the following delays last as long as set by `[fl_set_counter_min_repeat()], page 141`.

To determine the setting for "speedjumping" call

```
int fl_get_counter_speedjump(FL_OBJECT *obj);
```

17.5.5 Counter Attributes

Never use `FL_NO_BOX` as the boxtype for a counter.

The first color argument (`col1`) t `[fl_set_object_color()], page 287` controls the color of the background of the counter, the second (`col2`) sets the color of the arrow buttons of the counter.

17.5.6 Remarks

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

17.6 Spinner Object

A spinner object is a combination of a (numerical) input field with two (touch) buttons that allow to increment or decrement the value in the (editable) input field. I.e., the user can change the spinners value by either editing the value of the input field or by using the up/down buttons shown beside the input field.

There are two types of spinner objects, one for integer and one for floating point values. You can set limits on the values that can be input and you can also set the amount of increment/decrement achieved when clicking on its buttons.

17.6.1 Adding Spinner Objects

To add a spinner to a form use

```
FL_OBJECT *fl_add_spinner(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 on the left of the spinner object.

17.6.2 Spinner Types

There are two types of spinners, one for input of integer and one for floating point values:

`FL_INT_SPINNER`
: A spinner that allows input of integer values.

`FL_FLOAT_SPINNER`
: A spinner that allows input of floating point values.

The way a spinner looks like depends on its dimensions. If it's at least as wide as it's high the two buttons are drawn above each other to the right of the input field (and are marked with and up and down pointing triangle), while when the object is higher than it's wide they are drawn beside each other and below the input field (and the markers are then left and right pointing arrows).

17.6.3 Spinner Interaction

The user can change the value of a spinner in two ways. She can either edit the value in the spinner directly (exactly the same as for an integer or floating point input object (Chapter 18 [Part III Input Objects], page 147) or by clicking on one of the buttons that will increment or decrement the value.

Per default the spinner object gets returned to the application (or the associated callback is called) whenever the value changed and the interaction seems to have ended. If you want it returned under different circumstances use the function

```
void fl_set_object_return(FL_OBJECT *obj, unsigned int when);
```

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

`[FL_RETURN_NONE], page 45`
: Never return or invoke callback.

`[FL_RETURN_END_CHANGED], page 45`
: Return or invoke callback at end of interaction (when either the input field loses the focus or one of the buttons was released) and the spinner's value changed during the interaction.

`[FL_RETURN_CHANGED], page 44`
: Return or invoke callback whenever the spinner's value changed. This is the default.

`[FL_RETURN_END], page 44`
: Return or invoke callback at end of interaction regardless of the spinner's value having changed or not.

`[FL_RETURN_ALWAYS], page 45`
: Return or invoke callback whenever the value changed or the interaction ended.

17.6.4 Other Spinner Routines

Probably the most often used spinner functions are

```
double fl_get_spinner_value(FL_OBJECT *obj );
double fl_set_spinner_value(FL_OBJECT *obj, double val);
```

The first one returns the value of a spinner. The type of the return value is a double for both integer and floating point spinners, so you have to convert it for integer spinners appropriately, e.g: using the `FL_nint()` macro, that converts a double to the nearest integer value.

You can set or retrieve the upper and lower limit the value a spinner can be set to using the functions

```
void fl_set_spinner_bounds(FL_OBJECT *obj, double min, double max);
void fl_get_spinner_bounds(FL_OBJECT *obj, double *min, double *max);
```

Since this function is to be used for integer as well as floating point spinner objects the `double` type values must be converted as necessary for `[FL_INT_SPINNER], page 142`.

The default limits are `-10000` and `10000`, but can be set to up to `INT_MIN` and `INT_MIN` for `[FL_INT_SPINNER], page 142`s and `-DBL_MAX` and `DBL_MAX` for `[FL_FLOAT_SPINNER], page 142`s.

To set or determine the step size by which a spinner will be incremented or decremented when one of the buttons is clicked on use

```
void fl_set_spinner_step(FL_OBJECT *obj, double step);
double fl_get_spinner_step(FL_OBJECT *obj);
```

The default step size is `1` for both `[FL_INT_SPINNER], page 142` and `[FL_FLOAT_SPINNER], page 142` objects.

For `[FL_FLOAT_SPINNER], page 142` objects you can set (or determine) how many digits after the decimal point are shown by using

```
void fl_set_spinner_precision(FL_OBJECT *obj, int prec);
int fl_get_spinner_precision(FL_OBJECT *obj);
```

This is per default set to 6 digits after the decimal point. The function for setting the precision has no effect on `[FL_INT_SPINNER], page 142` objects and the other one returns 0 for this type of spinners.

17.6.5 Spinner Attributes

Please don't change the boxtype from `[FL_NO_BOX], page 109`.

The label color and font can be set using the normal `[fl_set_object_lcolor()], page 288`, `[fl_set_object_lsize()], page 288` and `[fl_set_object_lstyle()], page 288` functions. The color of the input field of a spinner object can be set via using `[fl_set_object_color()], page 287` where the first color argument (`col1`) controls the color of the input field when it is not selected and the second (`col2`) is the color when selected.

Instead of creating a plethora of functions to influence all the other aspects of how the spinner is drawn (colors, font types etc.) the user is given direct access to the sub-objects of a spinner. To this end three functions exist:

```
FL_OBJECT *fl_get_spinner_input(FL_OBJECT *obj);
```

```
FL_OBJECT *fl_get_spinner_up_button(FL_OBJECT *obj);
FL_OBJECT *fl_get_spinner_down_button(FL_OBJECT *obj);
```

They return the addresses of the objects the spinner object is made up from, i.e., that of the input field and the buttons for increasing and decreasing the spinner's value. These then can be used to set or query the way the individual component objects are drawn. The addresses of these sub-objects shouldn't be used for any other purposes, especially their callback function may never be changed!

17.7 Thumbwheel Object

Thumbwheel is another valuator that can be useful for letting the user indicate a value between some fixed bounds. Both horizontal and vertical thumbwheels exist. They have a minimum, a maximum and a current value (all floating point values). The user can change the current value by rolling the wheel.

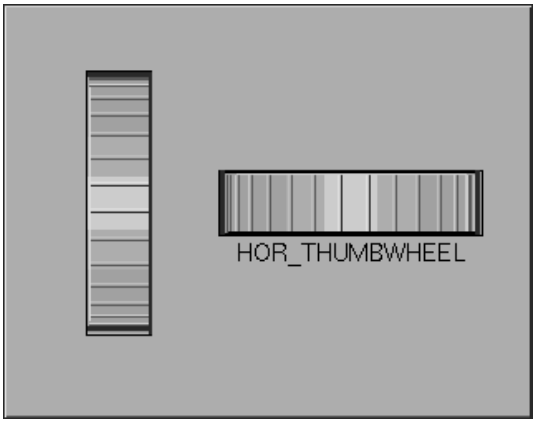
17.7.1 Adding Thumbwheel Objects

To add a thumbwheel to a form use

```
FL_OBJECT *fl_add_thumbwheel(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 thumbwheel.

17.7.2 Thumbwheel Types



The following types of thumbwheels are available:

`FL_VERT_THUMBWHEEL`
: A vertical thumbwheel.

`FL_HOR_THUMBWHEEL`
: A horizontal thumbwheel.

17.7.3 Thumbwheel Interaction

Whenever the user changes the value of the thumbwheel using the mouse or keyboard, the thumbwheel is returned (or the callback called) by the interaction routines. You change the

value of a thumbwheel by dragging the mouse inside the wheel area or, for vertical thumbwheels also by using the scroll wheel of the mouse. Each pixel of movement changes the value of the thumbwheel by 0.005, which you can change using the [fl_set_thumbwheel_step()], page 146 function.

The keyboard can be used to change the value of a thumbwheel. Specifically, the <Up> and <Down> cursor keys can be used to increment or decrement the value of a vertical thumbwheel and the <Right> and <Left> cursor keys can be used to increment or decrement the value of horizontal thumbwheel. Each pressing of the cursor key changes the thumbwheel value by the current step value. The <Home> key can be used to set the thumbwheel to a known value, which is the average of the minimum and the maximum value of the thumbwheel.

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

```
void fl_set_object_return(FL_OBJECT *obj, unsigned int when);
```

where the parameter when can be one of the four values

[FL_RETURN_NONE], page 45
: Never return or invoke callback.

[FL_RETURN_END_CHANGED], page 45
: Return or invoke callback at end (mouse release) if value is changed since last return.

[FL_RETURN_CHANGED], page 44
: Return or invoke callback whenever the thumbwheel value is changed.

[FL_RETURN_END], page 44
: Return or invoke callback at end (mouse release) regardless if the value is changed or not.

[FL_RETURN_ALWAYS], page 45
: Return or invoke callback whenever the value changes or the mouse button is released.

See demo program thumbwheel.c for an example use of this.

17.7.4 Other Thumbwheel Routines

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

```
double fl_set_thumbwheel_value(FL_OBJECT *obj, double val);
void fl_set_thumbwheel_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 value or bounds of a thumbwheel use

```
double fl_get_thumbwheel_value(FL_OBJECT *obj);
void fl_get_thumbwheel_bounds(FL_OBJECT *obj, double *min, double *max);
```

By default, the bounds are "hard", i.e., once you reach the minimum or maximum, the wheel would not turn further in this direction. However, if desired, you can make the bounds to turn over such that it crosses over from the minimum to the maximum value and vice versa. To this end, the following routine is available

```
int fl_set_thumbwheel_crossover(FL_OBJECT *obj, int yes_no);
```

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

```
void fl_set_thumbwheel_step(FL_OBJECT *obj, double step);
```

After this call thumbwheel values will be rounded to multiples of `step`. Use a value 0.0 for `step` to switch off rounding.

To get the current setting for this call

```
double fl_set_thumbwheel_step(FL_OBJECT *obj);
```

17.7.5 Thumbwheel Attributes

Setting colors via `[fl_set_object_color()], page 287` has no effect on thumbwheels.

17.7.6 Remarks

See the demo program `thumbwheel.c` for an example of the use of thumbwheels.

18 Input Objects

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.

18.1 Adding Input Objects

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.

18.2 Input 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 floating point numbers can be typed in (e.g., -23.2e12). The resulting string will be accepted by `strtod()` in its entirety (but may be too big to be represented by an `int` or `long`).

`FL_INT_INPUT`
: Only an integers can be typed in (e.g., -86). The resulting string will be accepted by `strtol()` in its entirety (but may be too big to be represented by an `float` or `double`).

`FL_DATE_INPUT`
: Only a date (MM/DD/YY or DD/MM/YY) can be typed in (and limited per default to 10 characters).

`FL_MULTILINE_INPUT`
: An input field allowing for multiple lines.

`FL_SECRET_INPUT`
: A normal input field that does not show the text (and limited per default to a maximum length of 16 characters).

`FL_HIDDEN_INPUT`
: A normal input field but invisible.

A normal input field can contain one line of text, to be typed in by the user. A float input field can only contain a float number. If the user tries to type in something else than a float, it is not shown and the bell is sounded. Similarly, an int input field can only contain an integer number and a date input field can only contain a valid date (see below). A multi-line input field can contain multiple lines of text. A secret input field works like a normal input field but the text is not shown (or scrambled). Only the cursor is shown which does move

while text is being entered. This can for example be used for getting passwords. Finally, a hidden input field is not shown at all but does collect text for the application program to use.

18.3 Input Interaction

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

delete previous char
: `<Backspace>`, `<Ctrl>h`

delete next char
: `<Delete>`

delete previous word
: `<Ctrl><Backspace>`

delete next word
: `<Ctrl><Delete>`

delete from cursor position to end of line
: `<Ctrl>k`

delete from cursor position to begin of line
: `<Meta>h`

jump to begin of line
: `<Ctrl>a`

jump to end of line
: `<Ctrl>e`

move char backward
: `<Ctrl>b`

move char forward
: `<Ctrl>f`

move to next line
: `<Ctrl>n`, `<Down>`

move to previous line
: `<Ctrl>p`, `<Up>`

move word backward
: `<Meta>b`

move word forward
: `<Meta>f`

move to begin of field
: `<Home>`

move to end of field
: `<End>`

clear input field
: `<Ctrl>u`

paste `<Ctrl>y`

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

A single click into the input field positions the cursor at the position of the mouse click.

There are three ways to select part of the input field. Dragging, double-click and triple-click. A double-click selects the word the mouse is on and a 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>` or `<Delete>` key or replaced by what the user types in.

One additional property of selecting part of the text field is that if the selection is done with the left mouse button 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 cut-buffers from other applications can be pasted into the input field. Use the middle mouse button for pasting. Note that `<Ctrl>y` only pastes the cut-buffer generated by `<Ctrl>k` and is not related to the X Selection mechanism, thus it only works within the same application. When the user presses the `<Tab>` key the input field is returned to the application program and the input focus is directed to the next input field. This also happens when the user presses the `<Return>` key but only if the form does not contain a return button. The order which input fields get the focus when the `<Tab>` is pressed is the same as the order the input fields were added to the form. From within Form Designer, using the raising function you can arrange (re-arrange) the focus order, see Section 10.6 [Raising and Lowering], page 90, in Part II for details. If the `<Shift>` key is pressed down when the `<Tab>` is pressed, the focus is directed to the previous input field.

Leaving an input field using the `<Return>`) key does not work for multi-line input fields since the `<Return>` key is used to start a new line.

Per default the input object gets returned to the application (or the callback set for the input object is invoked) when the input field is left and has been changed. Depending on the application, other options might be useful. To change the precise condition for the object to be returned (or its callback to become invoked), the following function can be used:

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

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

`[FL_RETURN_NONE], page 45`
: Never return or invoke callback

`[FL_RETURN_END_CHANGED], page 45`
: Default, object is returned or callback is called at the end if the field had been modified.

`[FL_RETURN_CHANGED], page 44`
: Return or invoke the callback function whenever the field had been changed.

`[FL_RETURN_END], page 44`
: Return or invoke the callback function at the end regardless if the field was modified or not.

[FL_RETURN_ALWAYS], page 45
: Return or invoke the callback function upon each keystroke and at the end (regardless if the field was changed or not)

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

A few additional notes: when you read "the fields has been changed" this includes the case that the user e.g., deleted a character and then added it back again. Also this case is reported as a "change" (a delete alone isn't) so the term "changed" does not necessarily mean that the content of the field has changed but that the user made changes (but which still might result in the exact same content as before).

Another term that may be understood differently is "end". In the versions since 1.0.91 it means that the users either hits the <Tab> or the <Return> key (except for multi-line inputs) or that she clicks onto some other object that in principle allows user interaction. These events are interpreted as an indication the user is done editing the input field and thus are reported back to the program, either by returning the input object or invoking its callback. But unless the user goes to a different input object the input field edited retains the focus.

Up to version 1.0.90 this was handled a bit differently: an "end of edit" event was not reported back to the program when the user clicked on a non-input object, i.e., changed to a different input object. This let to some problems when the interaction with the clicked-on non-input object dependet on the new content of the input object, just having been edited, but which hadn't been been reported back to the caller. On the other hand, some programs rely on the "old" behaviour. These programs can switch back to the traditional behaviour by calling the new function (available since 1.0.93)

```
fl_input_end_return_handling(int type);
```

where type can be either FL_INPUT_END_EVENT_ALWAYS, which is now the default, or FL_INPUT_END_EVENT_CLASSIC, which switches back to the type of handing used in versions up and including to 1.0.90. The function can be used at any time to change between the two possible types of behaviour. The function returns the previous setting.

There is a routine that can be used to limit the number of characters per line for input fields of type [FL_NORMAL_INPUT], page 147

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

To reset the limit to unlimited, set maxchars to 0. Note that input objects of type [FL_DATE_INPUT], page 147 are limited to 10 characters per default and those of type [FL_SECRET_INPUT], page 147 to 16.

Although an input of type [FL_RETURN_ALWAYS], page 45 can be used in combination with the callback function to check the validity of characters that are entered into the input field, use of the following method may simplify this task considerably:

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

The function filter() is called whenever a new (regular) character is entered. old is the string in the input field before the newly typed character c was added to form the new string 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. While the built-in filters also sound the keyboard bell, this don't happpens if a custom filter only returns FL_INVALID. To also sound the keyboard bell logically or it with FL_INVALID | FL_RINGBELL.

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

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

```
void fl_set_input_format(FL_OBJECT *obj, int attrib1, int attrib2);
void fl_get_input_format(FL_OBJECT *obj, int *attrib1, int *attrib2);
```

These two routines more or less provide a means for the validator to store and retrieve some information about user preference or other state dependent information. attrib1 and attrib2 can be any validator defined variables. For the built-in class, only the one of type [FL_DATE_INPUT], page 147 utilizes these to store the date format: for attrib1, it can take FL_INPUT_MMDD or FL_INPUT_DDMM and attrib2 is the separator between month and day. For example, to set the date format to dd/mm, use

```
fl_set_input_format(obj, FL_INPUT_DDMM, '/');
```

For the built-in type [FL_DATE_INPUT], page 147 the default is FL_INPUT_MMDD and the separator is '/'. There is no limit on the year other than it must be an integer and appear after month and day.

The function

```
int fl_validate_input(FL_OBJECT *obj);
```

can be used to test if the value in an input field is valid. It returns [FL_VALID], page 150 if the value is valid or if there is no validator function set for the input, otherwise [FL_INVALID], page 150.

There are two slightly different input modes for input objects. In the "normal" mode, when the input field is entered not using the mouse (e.g., by using of the <Tab> key) the cursor is placed again at the position it was when the field was left (or at the end of a possibly existing string when it's entered for the first time). When an input object has a maximum number of allowed characters set (via the [fl_set_input_maxchars()], page 150 function) and there's no room left no new input is accepted until at least one character has been deleted.

As an alternative there's an input mode that is similar to the way things were handle in DOS forms etc. Here, when the field is entered by any means but clicking into it with the mouse, the cursor is placed at the start of the text. And for fields with a maximum capacity, that contain already as many characters as possible, the character at the end of the field are removed when a new one is entered.

To switch between the two modes use the function

```
int fl_set_input_mode( int mode );
```

where `mode` is one of

`FL_NORMAL_INPUT_MODE`
: The default. Use it to switch to the "normal" input mode

`FL_DOS_INPUT_MODE`
: For selecting the DOS-like input mode

The function returns the previous setting. Note that the function changes the input mode for all input fields in your application.

18.4 Other Input Routines

Note that the label is not the default text in the input field. To set the contents of the input field use one of these routines:

```
void fl_set_input(FL_OBJECT *obj, const char *str);
void fl_set_input_f(FL_OBJECT *obj, const char *fmt, ...);
```

The first one takes a simple string while the second expects a format string with format specifiers just like `printf()` etc. and as many (appropriate) arguments as there are format specifiers.

Only a limited check on the string passed to the function is done in that only printable characters (according to the `isprint()` function) and, in the case of [FL_MULTILINE_INPUT], page 147 objects, new-lines (`'\n'`) are accepted (i.e., all that don't fit are skipped). Use an empty string (or a `NULL` pointer as the second argument) to clear an input field.

Setting the content of an input field does not trigger an object event, i.e., the object callback is not called. In some situations you might want to have the callback invoked. For this, you may use the function [fl_call_object_callback()], page 290.

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

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

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

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

```
void fl_set_input_selected(FL_OBJECT *obj, int flag);
void fl_set_input_selected_range(FL_OBJECT *obj, 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()], page 152 makes the entire input field selected. However, there is a subtle difference between this routine and [fl_set_input_selected()], page 152 when called with `flag` set to 1: [fl_set_input_selected()], page 152 always places the cursor at the end of the string while [fl_set_input_selected_range()], page 152q places the cursor at the beginning of the selection.

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

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

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

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

```
int fl_get_input_cursorpos(FL_OBJECT *obj, int *xpos, int *ypos);
```

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

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

```
void fl_set_input_cursorpos(FL_OBJECT *obj, int xpos, int ypos);
```

where `xpos` and `ypos` are measured in characters (lines). E.g., given the input field `"an arbitrary string"` the call

```
fl_set_input_cursorpos(ob, 4, 1);
```

places the the cursor after the first character of the word `"arbitrary"`, i.e., directly after the first `a`.

By default, if an input field of type `[FL_MULTILINE_INPUT], page 147` contains more text than can be shown, scrollbars will appear with which the user can scroll the text around horizontally or vertically. To change this default, use the following routines

```
void fl_set_input_hscrollbar(FL_OBJECT *obj, int how);
void fl_set_input_vscrollbar(FL_OBJECT *obj, int how);
```

where `how` can be one of the following values

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

`FL_ON` Always shows the scrollbar.

`FL_OFF` Never show scrollbar.

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

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

```
void fl_set_input_scroll(FL_OBJECT *obj, int yes_no);
```

There are also routines that can scroll the input field programmatically. To scroll vertically (for input fields of type [FL_MULTILINE_INPUT], page 147 only), use

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

where line is the new top line (starting from 1) in the input field. To programmatically scroll horizontally, use the following routine

```
void fl_set_input_xoffset(FL_OBJECT *obj, int pixels);
```

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

To obtain the current xoffset, use

```
int fl_get_input_xoffset(FL_OBJECT *obj);
```

It is possible to turn off the cursor of the input field using the following routine with a false value for yes_no:

```
void fl_set_input_cursor_visible(FL_OBJECT *obj, int yes_no);
```

To obtain the number of lines in the input field, call

```
int fl_get_input_numberoflines(FL_OBJECT *obj);
```

To obtain the current topline in the input field, use

```
int fl_get_input_topline(FL_OBJECT *obj);
```

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

```
int fl_get_input_screenlines(FL_OBJECT *obj);
```

For secret input field, the default is to draw the text using spaces. To change the character used to draw the text, the following function can be used

```
int fl_set_input_fieldchar(FL_OBJECT *obj, int field_char);
```

The function returns the old field char.

18.5 Input Attributes

Never use [FL_NO_BOX], page 109 as the boxtype.

The first color argument (col1) to [fl_set_object_color()], page 287 controls the color of the input field when it is not selected and the second (col2) 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, FL_COLOR tcol, FL_COLOR ccol);
```

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

If you want to know the colors of the text and cursor use

```
void fl_get_input_color(FL_OBJECT *obj, FL_COLOR *tcol, FL_COLOR *ccol);
```

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

```
void fl_set_input_scrollbarsize(FL_OBJECT *obj, int hh, int vw);
```

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

To determine the current settings for the horizontal scrollbar height and the vertical scrollbar width use

```
void fl_get_input_scrollbarsize(FL_OBJECT *obj, int *hh, int *vw);
```

The default scrollbar types are [FL_HOR_THIN_SCROLLBAR], page 131 and [FL_VERT_THIN_SCROLLBAR], page 131. There are two ways you can change the default. One way is to use [fl_set_defaults()], page 279 or [fl_set_scrollbar_type()], page 281 to set the application wide default (preferred); another way is to use [fl_get_object_component()], page 288 to get the object handle to the scrollbars and change the the object type forcibly. Although the second method of changing the scrollbar type is not recommended, the object handle obtained can be useful in changing the scrollbar colors etc.

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

```
typedef struct {
    long del_prev_char;     /* delete previous char */
    long del_next_char;     /* delete next char */
    long del_prev_word;     /* delete previous word */
    long del_next_word;     /* delete next word */
    long del_to_eol;        /* delete from cursor to end of line */
    long del_to_bol;        /* delete from cursor to begin of line */
    long clear_field;       /* delete all */
    long del_to_eos;        /* not implemented */
    long backspace;         /* alternative for del_prev_char */

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

    long transpose;         /* switch two char positions*/
    long paste;             /* paste the edit buffer */
} FL_EditKeymap;
```

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

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

with a filled or partially filled [FL_EditKeymap], page 155 structure. The unfilled members must be set to 0 so the default mapping is retained. Change of edit keymap is global and affects all input field within the application.

Calling [fl_set_input_editkeymap()], page 155 with km set to NULL restores the default. All cursor keys (<Left>, <Home> etc.) are reserved and their meanings hard-coded, thus

can't be used in the mapping. For example, if you try to set `del_prev_char` to `<Home>`, pressing the `<Home>` key will not delete the previous character.

To obtain the current map of the edit keys use the function

```
void fl_get_input_editkeymap(FL_EditKeymap *km);
```

with the `km` argument pointing of a user supplied structure which after the call will be set up with the current settings for the edit keys.

In filling the keymap structure, ASCII characters (i.e., characters with values below 128, including the control characters with values below 32) should be specified by their ASCII codes (`<Ctrl> C` is 3 etc.), while all others by their `Keysyms` (`XK_F1` etc.). Control and special character combinations can be obtained by adding `FL_CONTROL_MASK` to the `Keysym`. To specify `Meta` add `FL_ALT_MASK` to the key value.

```
FL_EditKeymap ekm;
memset(&ekm, 0, sizeof ekm);                  /* zero struct */

ekm.del_prev_char = 8;                        /* <Backspace> */
ekm.del_prev_word = 8 | FL_CONTROL_MASK;      /* <Ctrl><Backspace> */
ekm.del_next_char = 127;                      /* <Delete> */
ekm.del_prev_word = 'h' | FL_ALT_MASK;        /* <Meta>h */
ekm.del_next_word = 127 | FL_ALT_MASK;        /* <Meta><Delete> */
ekm.moveto_bof    = XK_F1;                    /* <F1> */
ekm.moveto_eof    = XK_F1 | FL_CONTROL_MASK;  /* <Ctrl><F1> */

fl_set_input_editkeymap(&ekm);
```

Note: In earlier versions of XForms (all version before 1.2) the default behaviour of the edit keys was slightly different which doesn't fit modern user expectations, as was the way the way the edit keymap was to be set up. If you use XForms for some older application that makes massive use of the "classical" behaviour you can compile XForms to use the old behaviour by using the `--enable-classic-editkeys` option when configuring the library for compilation.

18.6 Remarks

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

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

19 Choice Objects

19.1 Select Object

A select object is a rather simple object that allows the user to pick alternatives from a linear list that pops up when he clicks on the object. It remembers the last selected item, which is also shown on top of the select object.

The select object internally uses a popup (see Chapter 22 [Part III Popups], page 202) and thus it can be helpful to understand at lest some aspects of how popups work to fully grasp the functionality of select objects.

19.1.1 Adding Select Objects

To add a select object to a form use

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

There are currently three types which just differ by the way they look:

`FL_NORMAL_SELECT`
: Per default this type is drawn as a rounded, flat box (but you can change that by setting a different boxtype for the object) with the text of the currently selected item in its center.

`FL_MENU_SELECT`
: This select object looks like a button with a little extra box at its right side (just like a `FL_MENU_BUTTON`) and the text of the currently selected item is drawn on the button-like object.

`FL_DROPLIST_SELECT`
: This type looks like a button with the text of the currently selected item on top of it and a second square button directly beside it with an downward pointing arrow on it.

Per default `label` is drawn outside and to the left of the object.

Once a new select object has been created items have to be added to it. For this the following function exists:

```
FL_POPUP_ENTRY *fl_add_select_items(FL_OBJECT *obj,
                                    const char items,...);
```

`items` is a string with the items to add, separated by the `|` character. In the simplest case you would just use something like `"Item 1|Item 2|Item 3"` to add three items to the list. If there weren't any items before the first item will be automatically shown as the selected one.

As also described in the documentation for the similar function `[fl_popup_add_entries()], page 202` (see Section 22.1 [Adding Popups], page 202) the text for an item may contain "special sequences" that start with the character `%` and the may require an additional argument passed to the function after the `items` argument:

`%x`
: Set a value of type `long int` that's passed to all callback routines for the item. The value must be given in the arguments following the `items` string.

`%u` Set a `user_void` pointer that's passed to all callbacks of the item. The pointer must be specified in the arguments following the `items` string.

`%f` Set a callback function that gets called when the item is selected. The function is of type

```
int callback(FL_POPUP_RETURN *r);
```

Information about the item etc. gets passed to the callback function via the `[FL_POPUP_RETURN], page 206` structure and the return value of the function can be used to keep the selection from becoming reported back to the user made by returning a value of `FL_IGNORE` (-1). The function's address must be given in the arguments following the `items` string.

`%E` Set a callback routine that gets called each time the mouse enters the item (as long as the item isn't disabled or hidden). The type of the function is the same as that of the callback function for the selection of the item but it's return value is never used. The functions address must be given in the arguments following the `items` string.

`%L` Set a callback routine that gets called each time the mouse leaves the item. The type of the function is the same that as of the callback function for the selection of the item but it's return value is never used. The functions address must be given in the arguments following the `items` string.

`%d` Marks the item as disabled, i.e., it can't be selected and its text is per default drawn in a different color

`%h` Marks the item as hidden, i.e., it is not shown while in this state.

`%S` For items with shortcut keys it's quite common to have them shown on the right hand side. Using `"%S"` you can split the items text into two parts, the first one (before `"%S"`) being drawn flushed left and the second part flushed right. Note that using this special sequence doesn't automatically sets a shortcut key, this still has to be done using `"%s"`.

`%s` Sets one or more shortcut keys for an item. Requires a string with the shortcuts in the arguments following the `items` string. See Section 26.1 [Shortcuts], page 245, for details on how to define shortcuts. Please note that the character in the label identical to the shortcut character is only shown as underlined if `"%S"` isn't used.

`%%` Use this to get a `'%'` within the text of an item.

If you compare this list of "special sequences" with those listed for the `[fl_popup_add_entries()], page 202` function you will find that aome are missing. This is because a select object is a simple linear list of items that uses only parts of the popups functionalities.

Another way to set up the popup of a select object is to use the function

```
long fl_set_select_items(FL_OBJECT *obj, FL_POPUP_ITEM *item);
```

Here `item` is an array of structures of type `[FL_POPUP_ITEM], page 209` with the `text` member of the very last element of the array being set to `NULL`, indicating the end of the array.

The `text` member is the text of the item. It may only contain one "special sequence", `"%S"` to indicate that the string is to be split at that position into the part of the item label to be drawn to the left and on the right side (also prepending the string with `'_'` or `'/'` has no effect). `callback` is a callback function to be invoked on selection of the item. `shortcut` is a string for setting keybord shortcuts for the item. `type` has no function at all here (there can be only items of type `[FL_POPUP_NORMAL], page 207` in a select objects popup) and `state` can be set to `[FL_POPUP_DISABLED], page 207` and/or `[FL_POPUP_HIDDEN], page 207`.

Please note: when the select object already had items before the call of `[fl_set_select_items()], page 158` then they are removed before the new ones are set. The values assigned to the items start at 0.

A third way to "populate" a select object is to create a popup directly and then associate it with the select object using

```
int fl_set_select_popup(FL_OBJECT *obj, FL_POPUP *popup);
```

If the select object already had a popup before this will be deleted and replaced by the new popup passed as the second argument. Please note that the popup the argument `popup` points to may not contain any entries other than those of type `[FL_POPUP_NORMAL], page 207` (and, of course, the popup can't be a sub-popup of another popup).

19.1.2 Select Interaction

The simplest interaction with a select object consists of clicking onto the object and then selecting an item in the popup that gets shown directly beside the mouse position.

If you click with the left or right mouse button onto the select object previous or next item, respectively, will be selected. If youl keep the left or mouse button pressed down for a longer time slowly all alternatives are selected, one after each other.

You finally can also use the scroll wheel of your mouse to select the next or previous item (scrolling down selects the next, scrolling up the previous item).

On every selection of an item (also if the already selected item is re-selected) a callback that may have been associated with the item is executed. The callback receives as its argument a pointer to a structure of type `[FL_POPUP_RETURN], page 206`.

Its `val` member is a integer value associated with the entry. It can be set explicitely on creation of the item using the `"%x"` "special sequence". If not given then first item gets the value 0, the next 1 etc. `user_data` is a pointer to some user data, which can be set on creation of the item using `"%u"`. `text` is the string used in creating the item, including all "special sequences", while `label` is the string shown in the popup for the item. If there was a special sequence of `"%S"` in the string that was used to create the item `accel` is the text that appears right-flushed in the popup for the item. `entry` is a pointer to the popup entry that represents the item in the select object and, finally, `popup` is the popup associated with the select object.

Normally, when a new item is selected this is reported back to the caller either by calling the select objects callback (if one exists) or by returning the object as the result of a call of e.g., `[fl_do_forms()], page 296`. But if the callback for the item itself returns `FL_IGNORE` then the latter doesn't happen. This can be useful for cases where all work for a change of the selection can already be done within the items callback and the "main loop" shouldn't get involved anymore.

As for all other normal objects the condition under which a FL_SELECT object gets returned to the application (or an associate callback is called) can be influenced by calling the function

```
int fl_set_object_return(FL_OBJECT *obj, unsigned int when)
```

where when can have the following values

[FL_RETURN_NONE], page 45
: Never return or invoke a callback.

[FL_RETURN_END_CHANGED], page 45
: Return or invoke callback if end of interaction and selection of an item coincide.

[FL_RETURN_CHANGED], page 44
: Return or invoke callback whenever an item is selected (this is the default).

[FL_RETURN_END], page 44
: Return or invoke callback on end of an interaction.

[FL_RETURN_ALWAYS], page 45
: Return (or invoke callback) whenever the interaction ends and/or an item is selected.

Per default the popup of a select objects remains shown when the user releases the mouse somewhere outside the popup window (or on its title area). The alternative is to close the popup immediately when the user releases the mouse, independent of where it is. Using the function

```
int fl_set_select_policy(FL_OBJECT *obj, int policy);
```

the program can switch between these two modes of operation, where policy can be on of two values:

FL_POPUP_NORMAL_SELECT
: Keeps the popup opened when the mouse isn't released on one of the selectable items.

FL_POPUP_DRAG_SELECT
: Close the popup immediately when the mouse button is released.

The function returns on success the previous setting of the "policy" and -1 on error.

19.1.3 Other Select Routines

To find out which item is currently selected use

```
FL_POPUP_RETURN *fl_get_select_item(FL_OBJECT *obj);
```

It returns a pointer to a structure of type [FL_POPUP_RETURN], page 206 as already described above, containing all needed information about the selected item.

For some actions, e.g., deletion of an item etc., it is necessary to know the popup entry that represents it. Therefore it's possible to search the list of items according to several criteria:

```
FL_POPUP_ENTRY *fl_get_select_item_by_value(FL_OBJECT *obj, long val);
FL_POPUP_ENTRY *fl_get_select_item_by_label(FL_OBJECT *obj,
                                            const char *label);
FL_POPUP_ENTRY *fl_get_select_item_by_label_f(FL_OBJECT *obj,
```

```
                                                const char *fmt, ...);
FL_POPUP_ENTRY *fl_get_select_item_by_text(FL_OBJECT *obj,
                                           const char *text);
FL_POPUP_ENTRY *fl_get_select_item_by_text_f(FL_OBJECT *obj,
                                             const char *fmt, ...);
```

The first function, [fl_get_select_item_by_value()], page 160, searches through the list of items and returns the first one with the `val` associated with the item (or `NULL` if none is found). The second and third, [fl_get_select_item_by_label()], page 160 and [fl_get_select_item_by_label_f()], page 160 searches for a certain label as displayed for the item in the popup. The last two, [fl_get_select_item_by_text()], page 160 and [fl_get_select_item_by_text_f()], page 160 searches for the text the item was created by (that might be the same as the label text in simple cases). The difference between the second and third and the forth and the last is the way the text is passed to the functions, it's either a simple string or the result of the expansion of a format string as used for `printf()` etc. using the following unspecified arguments.

Please note that all these functions return a structure of type [FL_POPUP_ENTRY], page 206 (and not [FL_POPUP_RETURN], page 206, which gives you direct access to the entry in the popup for the item.

Using e.g., the result of one of the functions above you can also set the currently selected item via your program using

```
FL_POPUP_RETURN *fl_set_select_item(FL_OBJECT *obj,
                                    FL_POPUP_ENTRY *entry);
```

Or you could use the result to delete an item:

```
int fl_delete_select_item(FL_OBJECT *obj, FL_POPUP_ENTRY *entry);
```

Please note that the values associated with items won't change due to removing an item.

Alternatively, you can replace an item by one or more new ones. To do that use

```
FL_POPUP_ENTRY *fl_replace_select_item(FL_OBJECT *obj,
                                       FL_POPUP_ENTRY *old,
                                       const char *new_items, ...);
```

`old` designates the item to be removed and `new_items` is a string exactly like it would be used in [fl_add_select_items()], page 157 for the `items` argument, that defines the item(s) to replace the existing item. Please note that, unless values to be associated with the items (see the `val` member of the [FL_POPUP_RETURN], page 206 structure) there's a twist here. When items get created they per default receive increasing values, starting at 0. This also holds for items that get created in the process of replacement. The result is that the ordering of those values in that case wont represent the order in which they appear in the select objects popup.

Another sometimes useful function allows insertion of new items somewhere in the middle of a list of already existing items:

```
FL_POPUP_ENTRY *fl_insert_select_items(FL_OBJECT *obj,
                                       FL_POPUP_ENTRY *after,
                                       const char *new_items, ...);
```

`after` is the entry after which the new item(s) are to be inserted (if it's `NULL` the new items are inserted at the very start). The rest of the arguments are the same as for [fl_

replace_select_item()], page 161 and the same caveats about the values associated automatically with the new items holds.

It's possible to remove all items from a select object by calling

```
int fl_clear_select(FL_OBJECT *obj);
```

Afterwards you have to call again e.g., [fl_add_select_items()], page 157 to set new entries. Note that if you used [fl_set_select_popup()], page 159 to set a popup for the select object then that popup gets deleted automatically on calling [fl_clear_select()], page 162! The values automatically associated with items when calling [fl_add_select_items()], page 157 will start at 0 again.

19.1.4 Select Attributes

The two color arguments, clo1 and col2, of the function [fl_set_object_color()], page 287 set the background color of the object normally and when the mouse is hovering over it, respectively.

With the functions

```
FL_COLOR fl_set_selection_text_color(FL_OBJECT *obj, FL_COLOR color);
FL_COLOR fl_get_selection_text_color(FL_OBJECT *obj);
```

the color of the text of the currently selected item on top of the object can be set or queried.

To control (or determine) the alignment of the text with the currently selected item on top of the select object use

```
int fl_set_select_text_align(FLOBJECT *obj, int align);
int fl_get_select_text_align(FLOBJECT *obj);
```

Please note that the [FL_ALIGN_INSIDE], page 30 flag should be set with align since the text always will be drawn within the boundaries of the object. On success the function return the old setting for the alignment or -1 on error.

Finally, the font style and size of the text can be set or obtained using

```
int fl_set_select_text_font(FL_OBJECT *obj, int style, int size);
int fl_get_select_text_font(FL_OBJECT *obj, int *style, int *size);
```

The rest of the appearance of a select object concerns the popup that is used. To avoid bloating the API unnecessarily no functions for select objects were added that would just call popup functions. The popup belonging to a select object can be easily found from either a [FL_POPUP_ENTRY], page 206 structure as returned by the functions for searching for items or the [FL_POPUP_RETURN], page 206 structure passed to all callbacks and also returned by [fl_get_select_item()], page 160. Both structures have a member called popup that is a pointer to the popup associated with the select object. For popup functions operation on indiviual items just use the pointer to the [FL_POPUP_ENTRY], page 206 structure itself or the entry member of the [FL_POPUP_RETURN], page 206 structure.

There's also a convenience function for finding out the popup used for a select object:

```
FL_POPUP *fl_get_select_popup(FL_OBJECT *obj);
```

During the lifetime of a select object the popup never changes as long as [fl_set_select_popup()], page 159 isn't called.

Per default the popup of a select object does not have a title drawn on top of it. To change that use [fl_popup_set_title()], page 215.

To change the various colors and fonts used when drawing the popup use the functions [fl_popup_set_color()], page 216 and [fl_popup_entry_set_font()], page 216 (and [fl_popup_set_title_font()], page 215).

To change the border width or minimum width of the popup use [fl_popup_set_bw()], page 216 and [fl_popup_set_min_width()], page 216.

To disable or hide (or do the reverse) an item use the functions [fl_popup_entry_set_state()], page 214 and [fl_popup_entry_get_state()], page 214.

The keyboard shortcut for an entry can be set via [fl_popup_entry_set_shortcut()], page 217.

The callback functions (selection, enter and leave callback) for individual items can be set via [fl_popup_entry_set_callback()], page 213, [fl_popup_entry_set_enter_callback()], page 213 and [fl_popup_entry_set_leave_callback()], page 213, a callback for the whole popup with [fl_popup_set_callback()], page 213.

Finally, to assign a different (long) value to an item or set a pointer to user data use [fl_popup_entry_set_value()], page 217 and [fl_popup_entry_set_user_data()], page 217.

19.1.5 Remarks

See the demo program `select.c` for an example of the use of select objects.

19.2 Nmenu Object

Another object type that heavily depends on popups is the "nmenu" object type. It is meant to be used for menus and the "n" in front of the name stands for "new" since this is a re-implementation of the old menu object type (which is now deprecated since it is based on Section 23.3 [XPopup], page 227).

19.2.1 Adding Nmenu Objects

To add a nmenu object use

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

There are currently three types:

`FL_NORMAL_NMENU`
: Probably the most often used type: shown as text on a borderless background, popup gets opened when clicked on.

`FL_NORMAL_TOUCH_NMENU`
: Also shown as text on a borderless background, but popup gets opened when the mouse is moved on top of it without any further user action required.

`FL_BUTTON_NMENU`
: When not active shown as text on borderless background, when clicked on popup is shown and the object itself being displayed as a button.

`FL_BUTTON_TOUCH_NMENU`
: When not active shown as text on borderless background, when mouse is moved onto it the popup is shown and the object itself is displayed as a button.

Once a new nmenu object has been created items have to be added to it. For this the following function exists:

```
FL_POPUP_ENTRY *fl_add_nmenu_items(FL_OBJECT *obj,
                                   const char items, ...);
```

(The function can also be used to append new items to a nmenu object that already has items.)

The function returns a pointer to the first menu entry added on success and `NULL` on failure. `items` is a string with the items to add, separated by the `'|'` character. In the simplest case you would just use something like `"Item 1|Item 2|Item 3"` to add three items to the list.

As also described in the documentation for the similar function [fl_popup_add_entries()], page 202 the text for an item may contain "special sequences" that start with the character `'%'` and then may require an additional argument passed to the function after the `items` argument. All of those described in detail in the documentation for the [fl_popup_add_entries()], page 202 function can also be used for nmenus.

Another way to set up the popup of a select object, using an array of [FL_POPUP_ITEM], page 209, structures, is via the function

```
FL_POPUP_ENTRY *fl_set_nmenu_items(FL_OBJECT *obj, FL_POPUP_ITEM *item);
```

The function returns a pointer to the first menu item on success and `NULL` on failure. The function expects as arguments a pointer to the nmenu object and an array of [FL_POPUP_ITEM], page 209 structuress, with the very last element having `NULL` as the `text` member to mark the end of the array.

The `text` member of the structure may contain the character sequence `"%S"` to have the text drawn for the item split up at that position and with everything before `"%S"` drawn left-flushed and the rest right-flushed. Moreover, `text` may start with the character `'/'` and/or `'_'`. For an underline character a line is drawn above the item. And if there's a slash this item marks the begin of a sub-menu with all further items belonging to the sub-menu until a structure with member `text` being set to `NULL` is found in the array. (The `'/'` and `'_'` characters are, of course, not drawn.)

`type` indicates the type of the item. It can be

`FL_POPUP_NORMAL`
: A normal, plain item.

`FL_POPUP_TOGGLE`
: An item that represents one of two states and is drawn with a check-marker when in "on" state.

`FL_POPUP_RADIO`
: A radio item, i.e., it belongs to a group of items of which only one can be in "on" state at a time. They are drawn with a circle to the left with the circle for the "selected" item being filled with a color.

Please note that if `text` starts with a `'/'` the type **must** be `FL_POPUP_NORMAL`.

The `state` member per default is [FL_POPUP_NONE], page 207. It can be set to

`FL_POPUP_NONE`
: No special flags are set for the state of the item.

`FL_POPUP_DSABLED`
: The item is disabled and can't be selected.

`FL_POPUP_HIDDEN`
: The item is hidden, i.e., does not get shown (and thus can't be selected).

`FL_POPUP_CHECKED`
: Only relevant for toggle or radio items, marks it as in "on" state.

`callback` is a function that will be called if the item is selected. The callback function has the following type:

```
typedef int (*FL_POPUP_CB)(FL_POPUP_RETURN *);
```

It receives a pointer to a structure that contains all information about the entry selected by the user:

```
typedef struct {
    long int              val;       /* value assigned to entry */
    void                 *user_data; /* pointer to user data */
    const char           *text;      /* text of selected popup entry */
    const char           *label;     /* text drawn on left side */
    const char           *accel;     /* text drawn on right side */
    const FL_POPUP_ENTRY *entry;     /* selected popup entry */
    const FL_POPUP       *popup;     /* (sub-)popup it belongs to */
} FL_POPUP_RETURN;
```

`val` is a value that has been associated with the entry and `user_data` is a pointer that can be used to store the location of further information. `text` is the text that was used to create the entry (including all "special" characters), while `label` and `accel` are the texts shown for the entry on the left and right. `entry` is the pointer to the structure for the entry selected and `popup` to the (sub-) popup the entry belongs to (see Chapter 22 [Part III Popups], page 202, for more details on these structures).

If the callback function already does all the work required on selection of the item have it return the value `FL_IGNORE` to keep the selection from being reported back to the main loop of the program.

Finally, `shortcut` is a string encoding the keybord shortcut to be used for the item.

There's also a third method to "populate" a menu. If you already created a popup than you can set it as the menu's popup via a call of

```
int fl_set_nmenu_popup(FL_POPUP *popup);
```

Of course, the popup you associate with the nmenu object in this way can't be a sub-popup.

19.2.2 Nmenu Interaction

There are, if seen interaction-wise, two types of nmenu objects, normal ones and touch nmenus. For normal nmenus a popup is opened when the user clicks on the area of the nmenu object while for touch nmenus the popup already is shown when the user moves the mouse unto the area. In other respects they behave identical: the user just selects one of the items in the popup (or one of the sub-popups) and then the popup is closed again. The selection can now be handled within a callback function and/or reported back to the main loop of the program.

The popup is always shown directly below the nmenu object (except for the case that the popup is that long that it wouldn't fit on the screen, in which case the popup is drawn above the nmenu's area.

The most natural way to deal with a selection by the user is probably via a callback for the item that was selected. But also a callback for the popup as a whole or the object itself can be used. Item and popup callback functions are of type `[FL_POPUP_CB], page 205` described above (and in even more detail in Chapter 22 [Part III Popups], page 202), while object callbacks are "normal" XForms callback functions.

The condition under which a `FL_NMENU` object gets returned to the application (or an associate callback is invoked) can be influenced by calling the function

```
int fl_set_object_return(FL_OBJECT *obj, unsigned int when)
```

where `when` can have the following values

`[FL_RETURN_NONE], page 45`
: Never return or invoke a callback.

`[FL_RETURN_END_CHANGED], page 45`
: Return or invoke callback if end of interaction and selection of an item coincide.

`[FL_RETURN_CHANGED], page 44`
: Return or invoke callback whenever an item is selected (this is the default).

`[FL_RETURN_END], page 44`
: Return or invoke callback on end of an interaction.

`[FL_RETURN_ALWAYS], page 45`
: Return (or invoke callback) whenever the interaction ends and/or an item is selected.

One detail of the interaction that can be adjusted is under which conditions the nmenu's popup gets closed. Per default the popup is closed when an item is selected or (without a selection) when the user clicks somehwere outside of the popups area. This can be changed so that the popup also gets closed (without a selection) when the mouse button is clicked or released on a non-selectable item (giving the impression of a "pull-down" menu). For this purpose there's the

```
int fl_set_nmenu_policy(FL_OBJECT *obj, int policy);
```

function where `policy` can be one of two values:

`FL_POPUP_NORMAL_SELECT`
: Default, popup stays open until mouse button is released on a selectable entry or button is clicked outside the popups area.

`FL_POPUP_DRAG_SELECT`
: Popup is closed when the mouse button is released.

The function returns on success the previous setting of the "policy" and -1 on error.

19.2.3 Other Nmenu Routines

To find out which item of a nmenu object was selected last use

```
FL_POPUP_RETURN *fl_get_nmenu_item(FL_OBJECT *obj);
```

The function returns either a pointer to a [FL_POPUP_RETURN], page 206 structure with informations about the selected item (as already discussed above when talking about callbacks) or NULL if no selection was made the last time the nmenu object was used.

For some actions, e.g., deletion of an item etc., it is necessary to know the popup entry that represents it. Therefore it's possible to search the list of items according to several criteria:

```
FL_POPUP_ENTRY *fl_get_nmenu_item_by_value(FL_OBJECT *obj, long val);
FL_POPUP_ENTRY *fl_get_nmenu_item_by_label(FL_OBJECT *obj,
                                           const char *label);
FL_POPUP_ENTRY *fl_get_nmenu_item_by_label(FL_OBJECT *obj,
                                           const char *text);
```

The first function, [fl_get_nmenu_item_by_value()], page 167, searches through the list of all items (including items in sub-popups) and returns the first one with the val associated with the item (or NULL if none is found). The second, [fl_get_nmenu_item_by_label()], page 167 searches for a certain label as displayed for the item in the popup. The third, [fl_get_nmenu_item_by_text()], page 167 searches for the text the item was created by (that might be the same as the label text in simple cases). Please note that all functions return a structure of type [FL_POPUP_ENTRY], page 206 (and not [FL_POPUP_RETURN], page 206, which gives you direct access to the entry in the popup for the item.

Using e.g., the results of the above searches a nmenu item can be deleted:

```
int fl_delete_nmenu_item(FL_OBJECT *obj, FL_POPUP_ENTRY *item);
```

Alternatively, an item can be replaced by one or more items:

```
FL_POPUP_ENTRY *fl_replace_nmenu_item(FL_OBJECT *obj,
                                      FL_POPUP_ENTRY *old,
                                      const char *new_items, ...);
```

where old is the item to replace and new_items is a string exactly as used for [fl_add_nmenu_items()], page 164 with informations about the new item(s).

One also may insert additional items using

```
FL_POPUP_ENTRY *fl_insert_nmenu_items(FL_OBJECT *obj,
                                      FL_POPUP_ENTRY *after,
                                      const char *new_items, ...);
```

where after is the item after which the new items are to be inserted (use NULL to insert at the very start) and new_items is a string just like used with [fl_add_nmenu_items()], page 164 with informations about the additional item(s).

As you may remember, there are two different ways to "populate" a nmenu object. In one case you pass a kind of format string plus a variable number of arguments and in the other case an array of [FL_POPUP_ITEM], page 209 structures. The previously listed functions for inserting and replacing used the first "interface". But there are also three functions for using the alternative interface:

```
FL_POPUP_ENTRY *fl_add_nmenu_items2(FL_OBJECT *obj,
                                    FL_POPUP_ITEM *items);
FL_POPUP_ENTRY *fl_insert_nmenu_items2(FL_OBJECT *obj,
                                       FL_POPUP_ENTRY *after,
                                       FL_POPUP_ITEM *items);
```

```
FL_POPUP_ENTRY *fl_replace_nmenu_items2(FL_OBJECT *obj,
                                        FL_POPUP_ENTRY *old_item,
                                        FL_POPUP_ITEM *items);
```

All three functions return a pointer to the first new entry in the nmenu's popup on success and `NULL` on failure. The all take a pointer to the nmenu object as their first argument.

`[fl_add_nmenu_items2()], page 167` appends the items given by the list specified via the second argument to the nmenu's popup. `[fl_insert_nmenu_items2()], page 167` inserts one or more new items (as given by the last argument) after the entry specified by `after` (if `after` is `NULL` the new items are inserted before all existing items). Finally, `[fl_replace_nmenu_items2()], page 167` replaces the existing entry `old_item` with a new (or a list of new items specified by `items`.

Finally, there's a function to remove all items from a nmenu object at once:

```
in fl_clear_nmenu(FL_OBJECT *obj);
```

19.2.4 Nmenu Attributes

While not "active" the background of the nmenu object is drawn in the color that can be controlled via the first color argument, `col1`, of `[fl_set_object_color()], page 287`. When "active" (i.e., while the popup is shown) its background is drawn in the color of second color argument, `col2`, of the same function. The color of the label when "inactive" is controlled via `[fl_set_object_lcolor()], page 288`. When in "active" state the color use for the label can be set via the function

```
FL_COLOR fl_set_nmenu_hl_text_color(FL_OBJECT *obj, FL_COLOR color);
```

The function returns the old color on success or `[FL_MAX_COLORS], page 25` on failure. Per default this color is `FL_BLACK` for nmenus that are shown as a button while being "active" while for normal nmenus it's the same color that is used items in the popup when the mouse is hovering over them.

The size and style of the font used for the label of the nmenu object can be set via `[fl_set_object_lsize()], page 288` and `[fl_set_object_lstyle()], page 288`.

The rest of the appearance of a nmenu object is given by the appearance of the popup. These can be directly set via the functions for setting the popup appearance as described in Section 22.4 [Popup Attributes], page 215. To find out which popup is associated with the nmenu object use the function

```
FL_POPUP *fl_get_nmenu_popup(FL_OBJECT *obj);
```

and then use the popup specific functions to set the appearance. The same also holds for the appearance etc. of the items of the popup, a lot of functions exist that allow to set the attributes of entries of a popup, see Section 22.4 [Popup Attributes], page 215.

19.2.5 Remarks

See the demo program `menu.c`.

19.3 Browser Object

The browser object class 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 scrollbar 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 a (possible) long list of choices. Both single lines and multiple lines can be selected, depending on the type of the browser.

19.3.1 Adding Browser Objects

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.

19.3.2 Browser 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 type of browser the user can make single line selections which get reset immediately when the mouse button is released.

`FL_HOLD_BROWSER`
: Same as `FL_SELECT_BROSER` but the selection remains visible till the next selection.

`FL_DESELECTABLE_HOLD_BROWSER`
: Same as the `FL_HOLD_BROWSER` but the user can deselect the selected line.

`FL_MULTI_BROWSER`
: Multiple selections can be made and remain visible till de-selected.

Hence, the differences only lie in how the selection process works.

19.3.3 Browser Interaction

The user can change the position of the slider or use keyboard cursor keys (including `<Home>`, `<PageDown>`, etc.) to scroll through the text. When he/she presses the left mouse below or above the slider, the browser scrolls one page up or down. Any other mouse button scrolls one line at a time (except wheel mouse buttons). When not using an `[FL_NORMAL_BROWSER], page 169` the user can also make selections with the mouse by pointing to a line or by using the cursor keys.

For `[FL_SELECT_BROWSER], page 169`'s, as long as the user keeps the left mouse button pressed, the current line under the mouse is highlighted. Whenever she releases the left mouse button the highlighting disappears and the browser is returned to the application program. The application program can now figure out which line was selected using a call of `[fl_get_browser()], page 173` to be described below. It returns the number of the last selected line (with the topmost line being line 1).

A [FL_HOLD_BROWSER], page 169 works exactly the same except that, when the left mouse button is released, the selection remains highlighted. A [FL_DESELECTABLE_HOLD_BROWSER], page 169 additionally allows the user to undo a selection (by clicking on it again).

An [FL_MULTI_BROWSER], page 169 allows the user to select and de-select multiple lines. Whenever he selects or de-selects a line the browser object is returned to the application program (or a callback is executed when installed) that then can figure out which line was selected using [fl_get_browser()], page 173 (described in more detail below). That function returns the number of the last line to be selected or de-selected. When a line was de-selected the negation of the line number gets returned. I.e., if line 10 was selected the routine returns 10 and if line 10 was de-selected -10. When the user presses the left mouse button on a non-selected line and then moves it with the mouse button still pressed down, 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.

Per default a browser only gets returned (or a possibly associated callback gets invoked) on selection of a line (and, in the case of [FL_MULTI_BROWSER], page 169, on deselections). This behaviour can be changed by using the function

```
int fl_set_object_return(FL_OBJECT *obj, unsigned int when)
```

where `when` can have the following values

[FL_RETURN_NONE], page 45
: Never return or invoke callback.

[FL_RETURN_SELECTION], page 45
: Return or invoke callback on selection of a line. Please note that for [FL_MULTI_BROWSER], page 169 the browser may be returned just once for a number of lines having been selected.

[FL_RETURN_DESELECTION], page 45
: Return or invoke a callback on deselection of a line. This only happens for [FL_DESELECTABLE_HOLD_BROWSER], page 169 and [FL_MULTI_BROWSER], page 169 objects and, for the latter, the browser may get returned (or the callback invoked) just once for a number of lines having been deselected.

[FL_RETURN_END_CHANGED], page 45
: Return or invoke callback at end (mouse release) if the text in the browser has been scrolled.

[FL_RETURN_CHANGED], page 44
: Return or invoke callback whenever the text in the browser has been scrolled.

[FL_RETURN_END], page 44
: Return or invoke callback on end of an interaction for scrolling the text in the browser regardless if the text was scrolled or not.

[FL_RETURN_ALWAYS], page 45
: Return or invoke callback on selection, deselection or scrolling of text or end of scrolling.

The default setting for `when` for a browser object is `[FL_RETURN_SELECTION], page 45|[FL_RETURN_DESELECTION], page 45` (unless during the built of XForms you set the configuration flag `--enable-bwc-bs-hack` in which case the default is `[FL_RETURN_NONE], page 45` to keep backward compatibility with earlier releases of the library).

19.3.4 Other Browser Routines

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

To remove all lnes from a browser use

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

To add a line to a browser use one of

```
void fl_add_browser_line(FL_OBJECT *obj, const char *text);
void fl_add_browser_line_f(FL_OBJECT *obj, const char *fmt, ...);
```

The first function receives a simple string as the argument, the second one expects a format string just like for `printf()` etc. and followed by the appropriate number of arguments of the correct types. The line to be added may contain embedded newline characters (`'\n'`). These will result in the text being split up into several lines, separated at the newline characters.

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

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

The difference to `[fl_add_browser_line()], page 171` and `[fl_add_browser_line_f()], page 171` is that with these calls the browser will be shifted such that the newly appended line is visible. This is useful when e.g., using the browser to display messages.

Sometimes it may be more convenient to add characters to a browser without starting of a new line. To this end, the following routines exists

```
void fl_addto_browser_chars(FL_OBJECT *obj, const char *text);
void fl_addto_browser_chars_f(FL_OBJECT *obj, const char *fmt, ...);
```

These functions appends text to the last line in the browser without advancing the line counter. The to functions differ in that the first one takes a simple string argument while the second expects a format string just as for `printf()` etc., followed by a corresponding number of arguments. Again the text may contain embedded newline characters (`'\n'`). In that case, the text before the first embedded newline is appended to the last line, and everything afterwards is put onto new lines. As in the case of `[fl_addto_browser()], page 171` the last added line will be visible in the browser.

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

```
void fl_insert_browser_line(FL_OBJECT *obj, int line,
                            const char *text);
void fl_insert_browser_line_f(FL_OBJECT *obj, int line,
                              const char *fmt, ...);
```

The first function takes a simple string argument while the second one expects a format string as used for `printf()` etc. and the appropriate number of arguments (of the types specified in the format string).

Please note that on insertion (as well as replacements, see below) embedded newline characters don't result in the line being split up as it's done in the previous functions. Instead they will rather likely appear as strange looking characters in the text shown. The only exception is when inserting into an empty browser or after the last line, then this function works exactly as if you had called [fl_add_browser_line()], page 171 or [fl_add_browser_line_f()], page 171.

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 one of

```
void fl_replace_browser_line(FL_OBJECT *obj, int line,
                             const char *text);
void fl_replace_browser_line_f(FL_OBJECT *obj, int line,
                               const char *fmt, ...);
```

The first one takes a simple string for the replacement text while for the second it is to be specified by a format string exactly as used in printf() etc. and the appropriate number of arguments of the types specifed in the format string. \ As in the case of [fl_insert_browser_line()], page 171 and [fl_insert_browser_line_f()], page 171 newline characters embedded into the replacement text don't have any special meaning, i.e., they don't result in replacement of more than a single line.

Making many changes to a visible browser after another, e.g., clearing it and then adding a number of new lines, is slow because the browser is redrawn on each and every change. This can be avoided by using calls of [fl_freeze_form()], page 289 and [fl_unfreeze_form()], page 289. So a piece of code that fills in a visible browser should preferably look like the following

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

where browser->form is the form that contains the browser object named browser.

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 text of that line, exactly as it were passed to the function that created the line.

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

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

The routine returns 1 when file could be successfully loaded, otherwise 0. If the file name is an empty string (or the file could not be opened for reading) the browser is just cleared. This routine is particularly useful when using the browser for a help facility. You can create different help files and load the needed one depending on 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], page 169` are selected code similar to the following can be used:

```
int total_lines = fl_get_browser_maxline(browser);
for (i = 1; i <= total_lines; i++)
    if (fl_isselected_browser_line(browser, 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 *obj);
```

Please note that this count only includes lines that are shown completely in the browser, lines that are partially obscured aren't counted in.

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], page 169`) the negative of the de-selected line number is returned.

The following function allows to find out the (unobscured) line that is currently shown at the top of the browser:

```
int fl_get_browser_topline(FL_OBJECT *obj);
```

Note that the index of the top line is `1`, not `0`. A value of `0` is returned if the browser doesn't contain any lines.

Finally, the function

```
void fl_show_browser_line(FL_OBJECT *obj, int ln);
```

shifts the browsers content so that (as far as possible) the line indexed by `ln` is shown at the center of the browser.

It is possible to register a callback function that gets called when a line is double-clicked on. To do so, the following function is available:

```
void fl_set_browser_dblclick_callback(FL_OBJECT *obj,
                                      void (*cb)(FL_OBJECT *, long),
‘                                     long data);
```

Of course, double-click callbacks make most sense for [FL_HOLD_BROWSER], page 169s.

The part if the text visible within the browser can be set programmatically in a number of ways. With the functions

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

the line shown at the top or the bottom can be set (note again that line numbers start with 1).

Instead of by line number also the amount the text is scrolled in horizontal and vertical direction can be set with the functions

```
void fl_set_browser_xoffset(FL_OBJECT *obj, FL_Coord xoff);
void fl_set_browser_rel_xoffset(FL_OBJECT *obj, double xval);
void fl_set_browser_yoffset(FL_OBJECT *obj, FL_Coord yoff);
void fl_set_browser_rel_yoffset(FL_OBJECT *obj, double yval);
```

where `xoff` and `yoff` indicate how many pixels to scroll horizontally (relative to the left margin) or vertically (relative to the top of the text), while `xval` and `yval` stand for positions relative to the total width or height of all of the text and thus have to be numbers between `0.0` and `1.0`.

There are also a number of functions that can be used to obtain the current amount of scrolling:

```
FL_Coord fl_get_browser_xoffset(FL_OBJECT *obj);
FL_Coord fl_get_browser_rel_xoffset(FL_OBJECT *obj);
FL_Coord fl_get_browser_yoffset(FL_OBJECT *obj);
FL_Coord fl_get_browser_rel_yoffset(FL_OBJECT *obj);
```

Finally, there's a function that tells you the vertical position of a line in pixels:

```
int fl_get_browser_line_yoffset(FL_OBJECT *obj, imt line);
```

The return value is just the value that would have to be passed to [fl_set_browser_yoffset()], page 174 to make the line appear at the top of the browser. If the line does not exist it returns `-1` instead.

19.3.5 Browser Attributes

Never use the boxtype [FL_NO_BOX], page 109 for browsers.

The first color argument (`col1`) to [fl_set_object_color()], page 287 controls the color of the browser's box, the second (`col2`) the color of the selection. The text color is the same as the label color, `obj->lcol`.

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 [Label Attributes and Fonts], page 27, for details on font sizes and styles.

It is possible to change the appearance of individual lines in the browser. Whenever a line starts with the symbol `'@'` the next letter indicates the special characteristics 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	Italics modifier.
l	Large (new size is [FL_LARGE_SIZE], page 27).
m	Medium (new size is [FL_MEDIUM_SIZE], page 27).
s	Small (new size is [FL_SMALL_SIZE], page 27).
L	Large (new size = current size + 6)
M	Medium (new size = current size + 4)
S	Small (new size = current size - 2).
c	Centered.
r	Right aligned.
_	Draw underlined text.
-	An engraved separator. Text following '-' is ignored.
C	The next number indicates the color index for this line.
N	Non-selectable line (in selectable browsers).
' '	(a space character) Does nothing, can be used to separate between the digits specifying a color (following "@C", see above) and the text of a line starting with a digit.
@@	Regular '@' character.

The modifiers (bold and itatic) work by adding [FL_BOLD_STYLE], page 28 and [FL_ITALIC_STYLE], page 28 to the current active font index to look up the font in the font table (you can modify the table using [fl_set_font_name()], page 283 or [fl_set_font_name_f()], page 283).

More than one option can be used by putting them next to each other. For example, "@C1@l@f@b@cTitle" will give you the 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.

Depending on the font size and style lines may have different heights.

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 *obj, int key);
```

To align different text fields on a line, tab characters ('\t') can be embedded in the text. See [fl_set_tabstop()], page 282 on how to set tabstops.

There are two functions to turn the scrollbars on and off:

```
void fl_set_browser_hscrollbar(FL_OBJECT *obj, int how);
```

```
void fl_set_browser_vscrollbar(FL_OBJECT *obj, int how);
```

`how` can be set to the following values:

`FL_ON` Always on.

`FL_OFF` Always off.

`FL_AUTO` On only when needed (i.e., there are more lines/chars than could be shown at once in the browser).

`FL_AUTO` is the default.

Please note that when you switch the scrollbars off the text can't be scrolled by the user anymore at all (i.e., also not using methods that don't use scrollbars, e.g., using the cursor keys).

Sometimes, it may be desirable for the application to obtain the scrollbar positions when they change (e.g., to use the scrollbars of one browser to control other browsers). There are two ways to achieve this. You can use these functions:

```
typedef void (*FL_BROWSER_SCROLL_CALLBACK)(FL_OBJECT *, int, void *);
void fl_set_browser_hscroll_callback(FL_OBJECT *obj,
                                     FL_BROWSER_SCROLL_CALLBACK cb,
                                     void *cb_data);
void fl_set_browser_vscroll_callback(FL_OBJECT *obj,
                                     FL_BROWSER_SCROLL_CALLBACK cb,
                                     void *cb_data);
```

After scroll callbacks are set whenever the scrollbar changes position the callback function is called as

```
cb(ob, offset, cb_data);
```

The first argument to the callback function `cb` is the browser object, the second argument is the new xoffset for the horizontal scrollbar or the new top line for the vertical scrollbar. The third argument is the callback data specified as the third argument in the function calls to install the callback.

To uninstall a scroll callback, use a `NULL` pointer as the callback function.

As an alternative you could request that the browser object gets returned (or a callback invoked) when the the scrollbar positions are changed. This can be done e.g., by passing `[FL_RETURN_CHANGED], page 44` (if necessary `OR`'ed with flags for also returning on selection/deselections). Within the code for dealing with the event you could check if this is a change event by using the function

```
int fl_get_object_return_state(FL_OBJECT *obj);
```

and test if `[FL_RETURN_CHANGED], page 44` is set in the return value (by just logically `AND`'ing both) and then handle the change.

By default, the scrollbar size is based on the relation between the size of the browser and the size of the text. To change the default, use the following routine

```
void fl_set_browser_scrollbarsize(FL_OBJECT *obj, int hh, int vw);
```

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

The default scrollbar type is FL_THIN_SCROLLBAR. There are two ways you can change the default. One way is to use [fl_set_defaults()], page 279 or [fl_set_scrollbar_type()], page 281 to set the application wide default, another way is to use [fl_get_object_component()], page 288 to get the object handle to the scrollbars and change the the object type forcibly. The first method is preferable because the user can override the setting via resources. Although the second method of changing the scrollbar type is not recommended, the object handle obtained can be useful in changing the scrollbar colors etc.

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

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

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

Finally, the functions

```
int fl_get_browser_scrollbar_repeat(FL_OBJECT *obj);
void fl_set_browser_scrollbar_repeat(FL_OBJECT *obj, int millisec);
```

allows to determine and control the time delay (in milliseconds) between jumps of the scrollbar knob when the mouse button is kept pressed down on the scrollbar outside of the knobs area. The default value is 100 ms. The delay for the very first jump is twice that long in order to avoid jumping to start too soon when only a single click was intended but the user is a bit slow in releasing the mouse button.

19.3.6 Remarks

Since version 1.0.92 there isn't a limit on the maximum length of lines in a browser anymore. (The macro FL_BROWSER_LINELENGTH still exists and is set to 2048 for backward compatibility but has no function anymore).

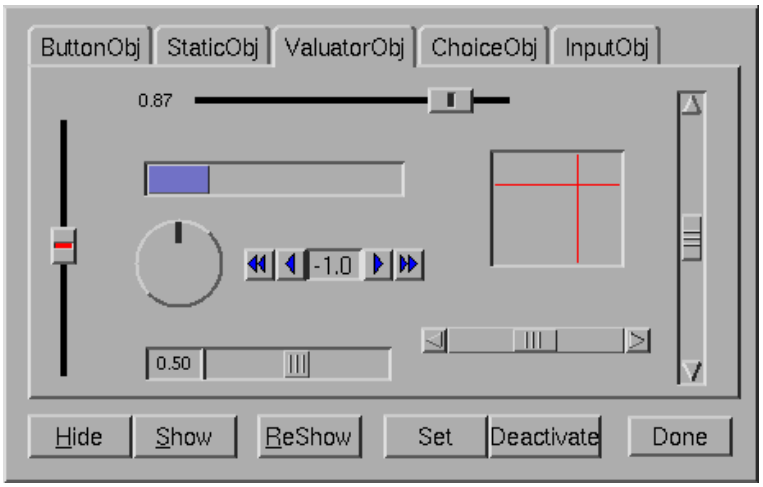
See fbrowse1.c for an example program using a [FL_NORMAL_BROWSER], page 169 to view files. browserall.c shows all different browsers. browserop.c shows the insertion and deletion of lines in a [FL_HOLD_BROWSER], page 169.

For the browser class, especially multi browsers, interaction via callbacks is strongly recommended.

20 Container Objects

20.1 Folder Object

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



20.1.1 Adding Folder Objects

To add a tabbed folder to a form use the routine

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

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

20.1.2 Folder Types

The following types are available:

`FL_TOP_TABFOLDER`
: Tabs on top of the folders.

`FL_BOTTOM_TABFOLDER`
: Tabs at the bottom of the folders.

20.1.3 Folder Interaction

The folders displayed by the tabbed folder class are simply regular forms (of type `FL_FORM`), which in turn contain objects. Each folder is associated with a name (shown on the tab rider). The folder interacts with the user just like any other form. Different from other top-level forms is that only one folder is active at any time. The user selects different folders by clicking on the tab rider associated with a folder.

To set up when the application is notified about events of the tabfolder or the tabfolders callback is invoked (if installed) use

```
void fl_set_object_return(FL_OBJECT *obj, unsigned int when);
```

where the `when` argument can be one of

`[FL_RETURN_NONE], page 45`
: Never report or invoke callback even if the selected folder has been changed.

`[FL_RETURN_CHANGED], page 44`
`[FL_RETURN_END_CHANGED], page 45`
: Result in a notification when a folder other that the currently active one has been selected (this is the default).

`[FL_RETURN_END], page 44`
`[FL_RETURN_ALWAYS], page 45`
: Notify when either a new or the already active folder has been selected.

In the releases before version 1.0.92 of the library only a callback for the folder was executed (if one was installed) on change of the selected folder bur not via e.g., `[fl_do_forms()], page 296` etc. This has changed with version 1.0.92. To get the old behaviour you have to build XForms with the `--enable-bwc-bs-hack` being set.

To find out which folder is currently active the following routines the tab riders are available

```
FL_FORM *fl_get_active_folder(FL_OBJECT *obj);
int fl_get_active_folder_number(FL_OBJECT *obj);
const char *fl_get_active_folder_name(FL_OBJECT *obj);
```

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

To find out what the previous active folder was (which may be of similar interest as the currently active one) the following functions can be used:

```
FL_FORM *fl_get_folder(FL_OBJECT *obj)
int fl_get_folder_number(FL_OBJECT *obj)
const char *fl_get_folder_name(FL_OBJECT *obj)
```

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

20.1.4 Other Folder Routines

To populate a tabbed folder, use the following routine

```
FL_OBJECT *fl_addto_tabfolder(FL_OBJECT *obj, const char *tab_name,
                              FL_FORM *folder)
```

where `tab_name` is a string (with possible embedded newlines in it) indicating the text of the tab rider and `folder` is a regular form created between calls of `[fl_bgn_form()], page 285` and `[fl_end_form()], page 285`. Only the pointer to the form is required. This means that the application program should not destroy a form that has been added to a

tabbed folder. The function returns the folder tab object, which is an object of class FL_BUTTON. The initial object color, label color, and other attributes (gravities, for example) of the tab button are inherited from the tabbed folder object obj and the location and size of the tab are determined automatically. You can change the attributes of the returned object just like any other objects, but not all possibilities result in a pleasing appearance. Note that although there is no specific requirement of what the backface of the folder/form should be, a boxtype other than FL_FLAT_BOX or FL_NO_BOX may not look nice. If the backface of the form is of FL_FLAT_BOX the associated tab will take on the color of the backface when activated.

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

To access the individual forms on the tabfolder, e.g., in order to modify something on it, use the following routines

```
FL_FORM *fl_get_tabfolder_folder_bynumber(FL_OBJECT *obj, int num);
FL_FORM *fl_get_tabfolder_folder_byname(FL_OBJECT *obj,
                                        const char *name);
FL_FORM *fl_get_tabfolder_folder_byname_f(FL_OBJECT *obj,
                                          const char *fnt, ...);
```

The functions take either the sequence number (the first tab on the left has a sequence number 1, the second 2 etc) or the tab name, which can either be passed directly as a string or via a format string like for printf() etc. and the corresponding (unspecified) arguments. The functions return the form associated with the number or the name. If the requested number or name is invalid, NULL is returned.

If there are more tabs than that can be shown, the right-most tab will be shown as "broken". Clicking on the "broken" tab scrolls the tab to the right one per each click. To scroll to the left (if there are tabs scrolled-off screen from the left), clicking on the first tab scrolls right. How many tabs are "hidden" on the left can be determined and also set using the functions

```
int fl_get_tabfolder_offset(FL_OBJECT *ojb);
int gl_set_tabfolder_offset(FL_OBJECT *obj, int offset);
```

where offset is the number of tabs hidden on the left.

Although a regular form (top-level) and a form used as a folder behave almost identically, there are some differences. In a top-level form, objects that do not have callbacks bound to them will be returned, when their states change, to the application program via [fl_do_forms()], page 296 or [fl_check_forms()], page 296. When a form is used as a folder, objects that do not have a callback are ignored even when their states changes. The reason for this behavior is that presumably the application does not care while the changes take place and they only become relevant when the the folder is switched off and at that time the application program can decide what to do with these objects' states (apply or ignore for example). If immediate reaction is desired, just use callback functions for these objects.

To obtain the number of folders in the tabfolder, the following routine can be used

```
int fl_get_tabfolder_numfolders(FL_OBJECT *obj);
```

To remove a folder, the following routine is available

```
void fl_delete_folder(FL_OBJECT *obj, FL_FORM *folder);
void fl_delete_folder_bynumber(FL_OBJECT *obj, int num);
void fl_delete_folder_byname(FL_OBJECT *obj, const char *name);
void fl_delete_folder_byname_f(FL_OBJECT *obj, const char *fmt, ...);
```

(the last two function differ in the way the tab names gets passed, the first is to be called with a simple string while the second expects a format string as used for `printf()` etc. and the appropriate number of arguments, from which the tab name gets constructed). wNote that after deletion, the number of folders in the tabfolder as well as the sequence numbers are updated. This means if you want to delete all folders after the second folder, you can do that by deleting the third folder repeatedly.

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

```
void fl_set_folder(FL_OBJECT *obj, FL_FORM *folder);
void fl_set_folder_bynumber(FL_OBJECT *obj, int num);
void fl_set_folder_byname(FL_OBJECT *obj, const char *name);
void fl_set_folder_byname_f(FL_OBJECT *obj, const char *fmt, ...);
```

(The latter two functions only differ in the way the tab name gets passed top them, the first accepts a simple string while the second expects a format string as used for `printf()` etc. and the appropriate number of (unspecified arguments, from which the tab name is constructed.)

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

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

where `x` and `y` are relative to the (top-level) form the tabbed folder belongs to. The size information may be useful for resizing the individual forms that has to go into the tabbed folder. Note that the folder area may not be constant depending on the current tabs (For example, adding a multi-line tab will reduce the area for the folders).

Since tab size can vary depending on monitor/font resolutions, it is in general not possible to design the forms (folders) so they fit exactly into the folder area. To dynamically adjust the sizes of the folders so they fit, the following routine is available

```
int fl_set_tabfolder_autofit(FL_OBJECT *obj, int how);
```

where `how` can be one of the following constants:

`FL_NO` Do not scale the form.

`FL_FIT` Always scale the form.

`FL_ENLARGE_ONLY`
Scale the form only if it is smaller than the folder area.

The function returns the old setting.

20.1.5 Remarks

By default, the tab for each folder is drawn with a corner of 3 pixels so it appears to be a trapezoid rather than a square. To change the appearance of the tabs, you can adjust the corner pixels using the following routine

```
int fl_set_default_tabfolder_corner(int n);
```

where `n` is the number of corner pixels. A value of 1 or 0 makes the tabs appear to be squarish. The function returns the old value.

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

Some visual effects like colors and label font of the tab rider buttons can be set all at once by calling the corresponding functions (i.e., `[fl_set_object_color()], page 287`, `[fl_set_object_lstyle()], page 288` etc.) with the tabbed folder object as the first argument. Individual tab rider buttons can also be modified by calling those function with the corresponding return value of `[fl_addto_tabfolder()], page 179` as the first argument.

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

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

A nested tabfolder might not work correctly at the moment.

20.2 FormBrowser Object

A form browser is another container class that is capable of holding multiple forms, the height of which in aggregate may exceed the screen height. The form browser also works obviously for a single form that has a height that is larger than the screen height.

This object class was developed with contributed code from Steve Lamont of UCSD and the National Center for Microscopy and Imaging Research (`spl@ucsd.edu`).

20.2.1 Adding FormBrowser Objects

Adding an object To add a formbrowser object to a form use the routine

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

The geometry indicated by `x`, `y`, `w` and `h` is the total area of the formbrowser, including scrollbars.

20.2.2 FormBrowser Types

There's only a single type of formbrowser available, the `FL_NORMAL_FORMBROWSER`.

20.2.3 FormBrowser Interaction

Once a formbrowser is populated with forms, you can scroll the forms with the scrollbars and interact with any of the forms. All objects on the forms act, for the most part, the same way as they would if they were on separate forms, i.e., if there are callback functions bound to the objects, they will be invoked by the main loop when the states of the objects change. However, objects on the form that do not have callbacks bound to them will not be returned by `[fl_do_forms()], page 296` or `[fl_check_forms()], page 296`.

Your application can be notified about changes of the scrollbars of the formbrowser. To set up under which conditions the application is notified or the formbrowsers callback is invoked (if installed) use

```
void fl_set_object_return(FL_OBJECT *obj, unsigned int when);
```

where the `when` argument can be one of

`[FL_RETURN_NONE], page 45`
: Never report or invoke callback (this is the default for the formbrowser object)

`[FL_RETURN_CHANGED], page 44`
: Result in a notification whenever the position of one of the scrollbars has changed.

`[FL_RETURN_END_CHANGED], page 45`
: Notification is sent if the position of a scrollbar has changed and the mouse button has been released.

`[FL_RETURN_END], page 44`
: Notification on release of the mouse button.

`[FL_RETURN_ALWAYS], page 45`
: Notify if the position of a scrollbar has changed or the mouse button has been released.

20.2.4 Other FormBrowser Routines

To populate a formbrowser, use the following routine

```
int fl_addto_formbrowser(FL_OBJECT *obj, FL_FORM *form);
```

where `form` is a pointer to a regular form created between calls of `[fl_bgn_form()], page 285` and `[fl_end_form()], page 285`. Only the form pointer is passed to the function, which means that the form should be valid for the duration of the formbrowser and the application program should not destroy a form that is added to a formbrowser before deleting the form from the formbrowser first. The function returns the total number of forms in the formbrowser. Note that although there is no specific requirement on what the backface of the form should be, not all boxtypes look nice.

The form so added is appended to the list of forms that are already in the formbrowser. You can also use the following routine to obtain the total number of forms in a formbrowser

```
int fl_get_formbrowser_numforms(FL_OBJECT *formbrowser);
```

Although a regular form (top-level) and a form used inside a formbrowser behave almost identically, there are some differences. In a top-level form, objects that do not have callbacks bound to them will be returned to the application program when their states change via `[fl_do_forms()], page 296` or `[fl_check_forms()], page 296`. When a form is used as member of a formbrowser those objects that do not have callbacks are ignored even when their states change.

To remove a form from the formbrowser, the following routine is available

```
int fl_delete_formbrowser(FL_OBJECT *obj, FL_FORM *form);
FL_FORM* fl_delete_formbrowser_bynumber(FL_OBJECT *obj, int num);
```

In the first function you specify the form to be removed from the formbrowser by a pointer to the form. If the form was removed successfully the function returns the remaining number of forms in the formbrowser, otherwise -1.

In the second function, you indicate the form to be removed with a sequence number, an integer between 1 and the number of forms in the browser. The sequence number is basically the order in which forms were added to the formbrowser. After a form is removed, the sequence numbers are re-adjusted so they are always consecutive. The function returns `NULL` if `num` was invalid, otherwise it returns address of the form that was removed.

To replace a form in formbrowser, the following routine is available

```
FL_FORM *fl_replace_formbrowser(FL_OBJECT *obj, int num,
                                FL_FORM *form);
```

where `num` is the sequence number of the form that is to be replaced by `form`. For example, to replace the first form in the browser with a different form, you should use 1 for `num`. The function returns the form that has been replaced on success, otherwise `NULL` is returned.

You can also insert a form into a formbrowser at arbitrary locations using the following routine

```
int fl_insert_formbrowser(FL_OBJECT *obj, int num, FL_FORM *form);
```

where `num` is the sequence number before which the new form `form` is to be inserted into the formbrowser. If successful the function returns the number of forms in the formbrowser, otherwise -1.

To find out the sequence number of a particular form, the following routine is available

```
int fl_find_formbrowser_form_number(FL_OBJECT *obj, FL_FORM *form);
```

The function returns a number between 1 and the number of forms in the formbrowser on success, otherwise 0.

To obtain the form handle from the sequence number, use the following routine

```
int fl_get_formbrowser_form(FL_OBJECT *obj, int num);
```

By default, if the size of the forms exceeds the size of the formbrowser, scrollbars are added automatically. You can use the following routines to control the scrollbars

```
void fl_set_formbrowser_hscrollbar(FL_OBJECT *obj, int how);
void fl_set_formbrowser_vscrollbar(FL_OBJECT *obj, int how);
```

where `how` can be one of the following

`FL_ON` Always on.

`FL_OFF` Always off.

`FL_AUTO` On when needed. This is the default.

The vertical scrollbar by default scrolls a fixed number of pixels. To change it so each action of the scrollbar scrolls to the next forms, the following routine is available

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

where `how` can be one of the following

`FL_SMOOTH_SCROLL`
The default.

`FL_JUMP_SCROLL`
: Scrolls in form increments.

To obtain the form that is currently the first form in the formbrowser visible to the user, the following can be used

```
FL_FORM *fl_get_formbrowser_topform(FL_OBJECT *obj);
```

You can also set which form to show by setting the top form using the following routine

```
int fl_set_formbrowser_topform(FL_OBJECT *obj, FL_FORM *form);
FL_FORM* fl_set_formbrowser_topform_bynumber(FL_OBJECT *obj, int num);
```

The first function returns the sequence number of the form and the second function returns the form with sequence number `num`.

Since the area occupied by the formbrowser contains the space for the scrollbars, the following routine is available to obtain the actual size of the forms area

```
void fl_get_formbrowser_area(FL_OBJECT *obj, int *x, int *y,
                             int *w, int *h);
```

where `x` and `y` are relative to the (top-level) form the formbrowser belongs to.

To programatically scroll within a formbrowser in horizontal and vertical direction, the following routines are available

```
int fl_set_formbrowser_xoffset(FL_OBJECT *obj, int offset);
int fl_set_formbrowser_yoffset(FL_OBJECT *obj, int offset);
```

where `offset` is a positive number, measuring in pixels the offset from the the natural position from the left and the top, respectively. In other words, 0 indicates the natural position of the content within the formbrowser. An x-offset of 10 means the content is scrolled 10 pixels to the left. Similarly an y-offset of 10 means the content is scrolled by 10 pixels upwards.

To obtain the current offsets, use the following routines

```
int fl_get_formbrowser_xoffset(FL_OBJECT *obj);
int fl_get_formbrowser_yoffset(FL_OBJECT *obj);
```

20.2.5 Remarks

A call of `fl_free_object(formbrowser)` does not free the individual forms, it only frees the formbrowser object itself.

See the demo program `formbrowser.c` for an example use of formbrowser class. A nested formbrowser might not work correctly at the moment.

21 Other Objects

21.1 Timer Object

Timer objects can be used to make a timer that runs down toward 0 or runs up toward a pre-set value after which it starts blinking and returns itself to the application program. This can be used in many different ways, for example, to give a user a certain amount of time for completing a task, etc. Also hidden timer objects 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 certain amount of time has passed.

The precision of the timer is not very high. Don't count on anything better than, say, 50 milli-seconds. Run demo `timerprec.c` for an actual accuracy measurement.

21.1.1 Adding Timer Objects

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.

21.1.2 Timer Types

There are at the moment three types of timers:

`FL_NORMAL_TIMER`
: Visible, Shows a label in a box which blinks when the timer expires.

`FL_VALUE_TIMER`
: Visible, showing the time left or the elapsed time. Blinks if the timer expires.

`FL_HIDDEN_TIMER`
: Not visible.

21.1.3 Timer Interaction

When a visible timer expires it starts blinking. The user can stop the blinking by pressing the mouse on it or by resetting the timer to 0.

The timer object is returned to the application program or its callback called when the timer expired per default. You can also switch off reporting the expiry of the timer by calling

```
int fl_set_object_return(FL_OBJECT *obj, unsigned int when)
```

with `when` set to `[FL_RETURN_NONE], page 45`. To re-enable reporting call it with one of `[FL_RETURN_CHANGED], page 44`, `[FL_RETURN_END], page 44`, `[FL_RETURN_END_CHANGED], page 45` or `[FL_RETURN_ALWAYS], page 45`.

21.1.4 Other Timer Routines

To set the timer to a particular value use

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

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

To obtain the time left in the timer use

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

By default, a timer counts down toward zero and the value shown (for `FL_VALUE_TIMER`s) is the time left until the timer expires. You can change this default so the timer counts up and shows elapsed time by calling

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

with a true value for the argument `yes_no`.

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

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

and later be resumed by

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

Unlike `[fl_set_timer()], page 186` a suspended timer keeps its internal state (total delay, time left etc.), so when it is resumed, it starts from where it was suspended.

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

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

The function `filter` receives the timer ID and the time left for count-down timers and the elapsed time for up-counting timers (in units of seconds) and should return a string representation of the time. The default filter returns the time in a `hour:minutes:seconds.fraction` format.

21.1.5 Timer Attributes

Never use `FL_NO_BOX` as the boxtype for `FL_VALUE_TIMER`s.

The first color argument (`col1`) to `[fl_set_object_color()], page 287` controls the color of the timer, the second (`col2`) is the blinking color.

21.1.6 Remarks

Although having different APIs and the appearance of a different interaction behaviour, the way timers and timeout callbacks work is almost identical with one exception: you can deactivate a timer by deactivating the form it belongs to. While the form is deactivated, the timers callback will not be called, even if it expires. The interaction will only resume when the form is activated again.

See `timer.c` for the use of timers.

21.2 XYPlot Object

A 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.

21.2.1 Adding XYPlot Objects

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 per default below it.

21.2.2 XYPlot Types

The following types are available:

`FL_NORMAL_XYPLOT`
: A solid line is drawn through the data points.

`FL_SQUARE_XYPLOT`
: Data drawn as a solid line plus squares at data points.

`FL_CIRCLE_XYPLOT`
: Data drawn as a solid line plus circles at data points.

`FL_FILL_XYPLOT`
: Data drawn as a solid line with the area under the curve filled.

`FL_POINTS_XYPLOT`
: Only data points are drawn with. per default, stars.

`FL_LINEPOINTS_XYPLOT`
: Data drawn as a solid line plus, per default, stars at data point.

`FL_DASHED_XYPLOT`
: Data drawn as a dashed line.

`FL_DOTTED_XYPLOT`
: Data drawn as a dotted line.

`FL_DOTDASHED_XYPLOT`
: Data drawn as a dash-dot-dash line.

`FL_IMPULSE_XYPLOT`
: Data drawn by vertical lines.

`FL_ACTIVE_XYPLOT`
: Data drawn as a solid line plus squares at data points, accepting manipulations.

`FL_EMPTY_XYPLOT`
: Only the axes are drawn.

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

XYPlots of type `FL_POINTS_XYPLOT` and `FL_LINEPOINTS_XYPLOT` are special in that the application can change the symbol drawn on the data point.

21.2.3 XYPlot Interaction

Only FL_ACTIVE_XYPLOT report mouse events by default. Clicking and dragging the data points (marked with little squares) will change the data and result in the object getting returned to the application (or the object's callback getting invoked). By default, the reporting happens only when the mouse is released. In some situations, reporting changes as soon as they happen might be desirable. To control when mouse events are returned use the function

```
int fl_set_object_return(FL_OBJECT *obj, unsigned int when);
```

where when can have the folowing values:

[FL_RETURN_NONE], page 45
: Never return or invoke callback.

[FL_RETURN_END_CHANGED], page 45
: Return or invoke callback at end (mouse release) if one of the points has been moved to a different place. This is the default.

[FL_RETURN_CHANGED], page 44
: Return or invoke callback whenever a point has been moved.

[FL_RETURN_END], page 44
: Return or invoke callback at end (mouse release) regardless if a point has been moved is changed or not.

[FL_RETURN_ALWAYS], page 45
: Return or invoke callback when a point has been moved or the mouse button has been release).

Please note: an object can also be in inspect mode (see function [fl_set_xyplot_inspect()], page 190 below). In this case the object gets returned (or its callback invoked) for all of the above settings except ([FL_RETURN_NONE], page 45) when the mouse was released on top of one of the points.

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

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

where via i the data index (starting from 0) is returned while via x and y the actual data point gets returned. If no point has changed i will be set to -1.

It is possible tp switch drawing of the squares that mark an active plot on and off (default is on) using the following routine

```
void fl_set_xyplot_mark_active(FL_OBJECT *obj, int yes_no);
```

with yes_no being set to false (0).

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);
void fl_set_xyplot_data_double(FL_OBJECT *obj, double *x, double *y, int n,
                               const char *title, const char *xlabel,
                               const char *ylabel);
```

(The `fl_set_xyplot_data_double()` function allows to pass data of type `double` but which get "demoted" to `float` type when assigned to the xyplot object.) Here `x`, `y` is the tabulated function, and `n` is the number of data points. If the xyplot object being set already exists old data will be cleared. Note that the tabulated function is copied internally so you can free or do whatever you want with the `x` and `y` arrays after the function has returned. `title` is a title that is drawn above the XYPlot and `xlabel` and `ylabel` are the labels drawn at the x- and y-axes.

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

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

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

To get a copy of the current XYPLot data, use

```
int fl_get_xyplot_data_size(FL_OBJECT *obj);
void fl_get_xyplot_data(FL_OBJECT *obj, float *x, *float y, int *n);
```

The first function returns the number of data points which the second will return. The caller must supply the space for the data returned by `fl_get_xyplot_data()`. The last argument of that function is again the number of points that got returned.

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

```
void fl_set_xyplot_inspect(FL_OBJECT *obj, int yes_no);
```

Once an XYPlot is in inspect mode, whenever the mouse is released and the mouse position is on one of the data point, the object is returned to the caller or its callback is invoked. You then can use `[fl_get_xyplot()], page 189` to find out which point the mouse was clicked on.

Another, perhaps even more general, way to obtain the values from an XYPlot is to use a posthandler or an overlay positioner. See demo `xyplotall.c` for the use of posthandler and `positionerXOR.c` for an example of reading-out xyplot values using an overlayed positioner.

21.2.4 Other XYPlot 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 *obj, int major, int minor);
void fl_set_xyplot_ytics(FL_OBJECT *obj, int major, int minor);
```

where `major` and `minor` are the number of tic marks to be placed on the axis and the number of divisions between major tic marks. In particular, -1 suppresses the tic marks completely while 0 restores the default settings (which is 5 for the major and 2 for the minor tic arguments).

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` and `minor` are only taken as a

hint to the scaling routine. However, in almost all cases the scaling routine will not generate a major tic that differs from the requested value by more than 3.

Normally the minor tics of logarithmic scales are drawn equidistant. To have them also drawn logarithmically use the functions

```
int fl_set_xyplot_log_minor_xtics(FL_OBJECT *obj, int yesno);
int fl_set_xyplot_log_minor_ytics(FL_OBJECT *obj, int yesno);
```

With this enabled e.g., the minor tics between 1 and 10 (when the interval is to be divided into 5 subintervals) will be drawn at the positions 2, 4, 6, and 8 instead of at 10^0.2, 10^0.4, 10^0.6 and 10^0.8. The functions return the previous setting.

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

```
void fl_set_xyplot_alphaxtics(FL_OBJECT *obj, const char *major,
                              const char *minor);
void fl_set_xyplot_alphaytics(FL_OBJECT *obj, const char *major,
                              const char *minor);
```

where `major` is a string specifying the labels with the embedded character `|` that specifies major divisions. For example, to label a plot with Monday, Tuesday etc, `major` should be given as `"Monday|Tuesday|..."`.

Parameter `minor` is currently unused and the minor divisions are set to 1, i.e, no divisions between major tic marks. Naturally the number of major/minor divisions set by this routine and `[fl_set_xyplot_xtics()], page 190` and `[fl_set_xyplot_ytics()], page 190` can't be active at the same time and the one that gets used is the one that was set last.

The above two functions can also be used to specify non-uniform and arbitary major divisions. To achieve this you must embed the major tic location information in the alphanumerical text. The location information is introduced by the `@` symbol and followed by a float or integer number specifying the coordinates in world coordinates. The entire location info should follow the label. For example, `"Begin@1|3/4@0.75|1.9@1.9"` will produce three major tic marks at 0.75, 1.0, and 1.9 with labels "3/4", "begin" and "1.9".

To get a gridded XYPlot use the following routines

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

where `xgrid` and `ygrid` can be one of the following

`FL_GRID_NONE`
: No grid.

`FL_GRID_MAJOR`
: Grid for the major divisions only.

`FL_GRID_MINOR`
: Grid for both the major and minor divisions.

The grid line by default is drawn using a dotted line, which you can change using the routine

```
int fl_set_xyplot_grid_linestyle(FL_OBJECT *obj, int style);
```

where `style` is the line style (`FL_SOLID`, `FL_DASH` etc. See Chapter 28 [Drawing Objects], page 254, for a complete list). The function returns the old grid linestyle.

By default, the plotting area is automatically adjusted for tic labels and titles so that a maximum plotting area results. This can in certain situations be undesirable. To control the plotting area manually, the following routines can be used

```
void fl_set_xyplot_fixed_xaxis(FL_OBJECT *obj, const char *lm,
                               const char *rm)
void fl_set_xyplot_fixed_yaxis(FL_OBJECT *obj, const char *bm,
                               const char *tm)
```

where `lm` and `rm` specify the right and left margin, respectively, and `bm` and `tm` the bottom and top margins. The pixel amounts are computed using the current label font and size. Note that even for y-axis margins the length of the string, not the height, is used as the margin, thus to leave space for one line of text, a single character (say `m`) or two narrow characters (say `ii`) should be used.

To restore automatic margin computation, set all margins to `NULL`.

To change the size of the symbols drawn at data points, use the following routine

```
void fl_set_xyplot_symbolsize(FL_OBJECT *obj, int size);
```

where `size` should be given in pixels. The default is 4.

For `FL_POINTS_XYPLOT` and `FL_LINEPOINTS_XYPLOT` (main plot or overlay), the application program can change the symbol using the following routine

```
typedef void (*FL_XYPLOT_SYMBOL)(FL_OBJECT *, int id,
                                 FL_POINT *p, int n, int w, int h);
FL_XYPLOT_SYMBOL fl_set_xyplot_symbol(FL_OBJECT *obj, int id,
                                      FL_XYPLOT_SYMBOL symbol);
```

where `id` is the overlay id (0 means the main plot, and you can use -1 to indicate all), and `symbol` is a pointer to the function that will be called to draw the symbols on the data point. The parameters passed to this function are the object pointer, the overlay `id`, the center of the symbol (`p->x`, `p->y`), the number of data points (`n`) and the preferred symbol size (`w`, `h`). If the type of the XYPlot corresponding to `id` is not `FL_POINTS_XYPLOT` or `FL_LINESPOINTS_XYPLOT`, the function will not be called.

To change for example a `FL_LINEPOINTS_XYPLOT` XYPlot to plot filled small circles instead of the default crosses, the following code could be used

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

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

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

If a Xlib drawing routine is used it should use the current active window (FL_ObjWin(obj)) and the current GC. Take care not to call routines inside the drawsymbol() function that could trigger a redraw of the XYPlot (such as [fl_set_object_color()], page 287, [fl_set_xyplot_data()], page 189 etc.).

To use absolute bounds (as opposed to the bounds derived from the data), use the following routines

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

Data that fall outside of the range set this way will be clipped. To restore autoscaling, call the function with max and min set to exactly the same value. To reverse the axes (e.g., min at right and max at left), set min > max for that axis.

To get the current bounds, use the following routines

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

To replace the value of a particular point use the routine

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

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

It is possible to overlay several plots together by calling

```
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 (currently 32). Again, the data are copied to an internal buffer (old data are freed if necessary).

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

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

The function returns the number of data points successfully read. The type (FL_NORMAL_XYPLOT etc.) used in overlay plot is the same as the object itself.

To change an overlay style, use the following call

```
void fl_set_xyplot_overlay_type(FL_OBJECT *obj, int id, int type);
```

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

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

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

where id specifies the overlay number between 1 and FL_MAX_XYPLOTOVERLAY or the number set via [fl_set_xyplot_maxoverlays()], page 194 (see below). (Actually, when id is zero, this function returns the base data). The caller must supply the storage space for the data. Upon function return, n will be set to the number of data points retrieved.

Sometimes it may be more convenient and efficient to get the pointer to the data rather than a copy of the data. To this end, the following routine is available

```
void fl_get_xyplot_data_pointer(FL_OBJECT *obj, int id,
```

```
                                        float **x, float **y, int *n);
```

Upon function return, x and y are set to point to the data storage. You're free to modify the data and redraw the XYPlot (via [fl_redraw_object()], page 297). The pointers returned may not be freed.

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 *obj, int maxoverlays);
```

The function returns the previous maximum number of overlays.

To obtain the number of data points, use the routine

```
int fl_get_xyplot_numdata(FL_OBJECT *obj, int id);
```

where id is the overlay ID (with 0 being the base data set).

To insert a point into an xyplot, use the following routine

```
void fl_insert_xyplot_data(FL_OBJECT *obj, int id, int n,
                           double x, double y);
```

where id is the overlay ID; n is the index of the point after which the data new point specified by x and y is to be inserted. Set n to -1 to insert the point in front. To append to the data, set n to be equal or larger than the return value of fl_get_xyplot_numdata(obj, id).

To delete an overlay, use the following routine

```
void fl_delete_xyplot_overlay(FL_OBJECT *obj, int id);
```

It is possible to place inset texts on an XYPlot using the following routine (up to FL_MAX_XYPLOTOVERLAY or the value set via [fl_set_xyplot_maxoverlays()], page 194 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 (world) coordinates where text is to be placed and align specifies the placement options relative to the specified point (See [fl_set_object_lalign()], page 288 for valid options). If you for example specify FL_ALIGN_LEFT, the text will appear on the left of the point and flushed toward the point (see Fig. 21.1). This is mostly consistent with the label alignment except that now the bounding box (of the point) is of zero dimension. Normal text interpretation applies, i.e., if text starts with @ a symbol is drawn.

To remove an inset text, use the following routine

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

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

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

where id again is the overlay ID. To remove a key, set the key to NULL. All the keys will be drawn together inside a box. The position of the keys can be set via

```
void fl_set_xyplot_key_position(FL_OBJECT *obj, float x, float y,
```

```
                                        int align)
```

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

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

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

where `keys` specifies the keys for each plot. The last element of the array must be `NULL` to indicate the end. The array index is the plot id, i.e., `key[0]` is the key for the base plot, `key[1]` the key for the the first overlay etc.

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

```
void fl_set_xyplot_key_font(FL_OBJECT *obj, int style, int size);
```

Data may be interpolated using an nth order Lagrangian polynomial:

```
void fl_set_xyplot_interpolate(FL_OBJECT *obj, int id, int degree,
                               double grid);
```

where `id` is the overlay ID (use 0 for the base data set); `degree` is the order of the polynomial to use (between 2 and 7) and `grid` is the working grid onto which the data are to be interpolated. To restore the default linear interpolation, use `degree` set to 0 or 1.

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

```
void fl_set_xyplot_linewidth(FL_OBJECT *obj, int id, int width);
```

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

By default, a 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 *obj, int scale, double base);
void fl_set_xyplot_yscale(FL_OBJECT *obj, int scale, double base);
```

where the valid scaling options for scale are q`FL_LINEAR` and `FL_LOG`, and `base` is used only for `FL_LOG` and in that case is the base of the logarithm to be used.

Use the following routine to clear an xyplot

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

This routine frees all data associated with an XYPlot, including all overlays and all inset texts. This routine does not reset all plotting options, such as line thickness, major/minor divisions etc. nor does it free all memories associated with the XYPlot, for this `[fl_free_object()], page 286` is needed.

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

```
void fl_get_xyplot_xmapping(FL_OBJECT *obj, float *a, float *b);
void fl_get_xyplot_xmapping(FL_OBJECT *obj, float *a, float *b);
```

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

```
screenCoord = a * data + b                 (linear scale)
screenCoord = a * log(data) / log(p) + b   (log scale)
```

where p is the base of the requested logarithm.

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

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

Finally, there's a function for returning the coordinates of the area of the object used for drawing the data (i.e., the area, when axes are displayed, which is enclosed by the axes):

```
void fl_get_xyplot_screen_area(FL_OBJECT *obj,
                               FL_COORD *llx, FL_COORD *lly,
                               FL_COORD *urx, FL_COORD *ury);
void fl_get_xyplot_world_area(FL_OBJECT *obj,
                              float *llx, float *lly,
                              float *urx, float *ury);
```

where via llx and lly the coordinates of the lower left hand corner and via urx and ury those of the upper right hand corner are returned. The first function returns the corner positions in screen coordinates (relative to the object), while the secoind returns them in "world" coordinates.

21.2.5 XYPlot Attributes

Don't use FL_NO_BOX as the boxtype of an XYPlot object that is to be changed dynamically. To change the font size and style for the tic labels, inset text etc., use [fl_set_object_lsize()], page 288 and [fl_set_object_lstyle()], page 288.

The first color argument (col1) to [fl_set_object_color()], page 287 controls the color of the box and the second (col2) the actual XYPlot color.

21.2.6 Remarks

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

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

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

See xyplotall.c and xyplotactive.c for examples of the use of XYPlot objects. There is also an example program called xyplotover.c, which shows the use of overlays. In addition, xyplotall.c shows a way of getting all mouse clicks without necessarily using an active XYPlot.

It is possible to generate a PostScript output of an XYPlot. See the function [fl_object_ps_dump()], page 291 documented in Part V.

21.3 Canvas Object

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()], page 289` to control its position and size after its parent form is resized.

You also should be aware that when using a canvas you'll probably mostly program directly using basic Xlib functions, XForms doesn't supply much more than a few helper functions. You'll rather likely draw to it with Xlib functions and will be dealing with `XEvent`s yourself (instead having them taken care of by XForms and cenverted to some simpler to use events that then just return the object from `[fl_do_forms()], page 296` or invoke an associated callback function. Thus you will typically need a basic knowledge of how to program via the X11 Xlib.

21.3.1 Adding Canvas Objects

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, the window name will be generated on the fly as `flcanvasn`, where `n = 0, 1,...`.

21.3.2 Canvas Types

The only types of canvases currently available is `FL_NORMAL_CANVAS`.

21.3.3 Canvas Interaction

The canvas class is designed to maximize the programmer's ability to deal with situations where standard form classes may not be flexible enough. With canvases, the programmer has complete control over everything that can happen to a window. It thus doesn't work like other objects that get returned by `[fl_do_forms()], page 296` etc. or have their callbacks invoked.

Instead the user can request that for specific `X` events (not XForms object events like `FL_PRESS`, `FL_KEYPRESS` etc.!) callbacks are invoked that receive all information about the `XEvent` that led to their invocation. This obviously requires some understanding of how the X Window system works.

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

```
typedef int (*FL_HANDLE_CANVAS)(FL_OBJECT *obj, Window win,
                                int win_width, int win_height,
                                XEvent *xev, void *user_data);
void fl_add_canvas_handler(FL_OBJECT *obj, int event,
                           FL_HANDLE_CANVAS handler, void *user_data);
```

where `event` is the `XEvent` type, e.g., `Expose` etc. The `[fl_add_canvas_handler()], page 197` function first registers a procedure with the event dispatching system of the Forms Library, then it figures out the event masks corresponding to the event `event` and invokes `[fl_addto_selected_xevent()], page 53` to solicit the event from the server. Other book keeping (e.g., drawing the box that encloses the canvas, etc.) is done by the object handler.

When a canvas handler is installed the library tries to set the correct mask for the the `XEvent` (which then tells the X Window system which events to pass on to the Forms Library). But since translation from an `XEvent` to an `XEvent` mask is not unique, the default translation of the `XEvent` to a mask may or may not match exactly the intention of the application. Two events, namely `MotionNotify` and `ButtonPress`, are likely candidates that need further clarification from the application. There are two functions to add or delete from the mask, `[fl_addto_selected_xevent()], page 53` and `[fl_remove_selected_xevent()], page 53`.

By default, when a mouse motion handler (i.e., for the `MotionNotify` events) is registered, it is assumed that, while the application wants to be informed about mouse movements, it's not interested in a continous motion monitoring (tracking), thus per default `MotionNotify` events are requested with `PointerMotionHintMask` being set in the mask to reduce the number of events generated. If this is not the case and in fact the application wants to use the mouse motion as some type of graphics control, the default behavior would appear "jerky" as not every mouse motion is reported. To change the default behavior so that every mouse motion is reported, you need to call `[fl_remove_selected_xevent()], page 53` with mask set to `PointerMotionHintMask`. Furthermore, the mouse motion is reported regardless if a mouse button is pressed or not. If the application is interested in mouse motion only when a mouse button is pressed `[fl_remove_selected_xevent()], page 53` should be called with a mask of `PointerMotionMask|PointerMotionHintMask`.

With `ButtonPress` events you need to call `[fl_addto_selected_xevent()], page 53` with a mask of `OwnerGrabButtonMask` if you are to add or remove other canvas handlers in the button press handler.

To remove a registered handler, use

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

After this function call the canvas ceases to receive the events for `event`. The corresponding default bits in the `XEvent` mask as were set by `[fl_add_canvas_handler()], page 197` are cleared. If you added extra ones with `[fl_addto_selected_xevent()], page 53` you should reset them using `[fl_remove_selected_xevent()], page 53`.

To obtain the window ID of a canvas, use

```
Window fl_get_canvas_id(FL_OBJECT *obj);
```

or use the generic function (macro) (recommended)

```
Window FL_ObjWin(FL_OBJECT *obj);
```

Of course, the window ID only has a meaning after the form/canvas is shown. When the canvas or the form the canvas is on is hidden (via `[fl_hide_object()], page 289` or `[fl_hide_form()], page 295`), the canvas window may be destroyed. If the canvas is shown again, a new window ID for the canvas may be created. Thus recording the canvas window

ID in a static variable is not the right thing to do. It is much safer (and it doesn't add any run-time overhead) to obtain the canvas window ID via [FL_ObjWin()], page 198 whenever it's needed. If your application must show and hide the canvas/form repeatedly, you might consider to "unmap" the window, a way of removing the window from the screen without actually destroying it and later re-mapping the window to show it. The Xlib API functions for doing this are XUnmapWindow() and XMapWindow(). Both require two arguments. the display, which you can determine by calling [fl_get_display()], page 255 and the window ID, which can be obtained by using form->window if you want to (un)map a form or FL_ObjWin(obj) for a canvas.

21.3.4 Other Canvas Routines

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

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

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

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

```
void fl_set_canvas_colormap(FL_OBJECT *obj, Colormap map);
Colormap fl_get_canvas_colormap(FL_OBJECT *obj);
void fl_set_canvas_visual(FL_OBJECT *obj, Visual *vi);
void fl_set_canvas_depth(FL_OBJECT *obj, int depth);
int fl_get_canvas_depth(FL_OBJECT *obj);
```

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

Caution should also applied when using [fl_set_canvas_colormap()], page 199: when the canvas window goes away, e.g., as a result of a call of [fl_hide_form()], page 295, the colormap associated with the canvas is freed (destroyed). This likely will cause problems if a single colormap is used for multiple canvases as each canvas will attempt to free the same colormap, resulting in an X error. If your application works this way, i.e., the same colormap is used on multiple canvases (via [fl_set_canvas_colormap()], page 199), you should use the following routine to prevent the canvas from freeing the colormap:

```
void fl_share_canvas_colormap(FL_OBJECT *obj, Colormap colormap);
```

This function works the same way as [fl_set_canvas_colormap()], page 199 except that it also sets a internal flag so the colormap isn't freed when the canvas goes away.

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

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

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

where `n_colors` indicates how many colors in the newly created colormap should be filled with XForms' default colors (to avoid flashing effects). Note however, that the colormap entry 0 is allocated with either black or white even if you specify 0 for `n_colors`. To prevent this from happening (so you get a completely empty colormap), set `n_colors` to -1. See Chapter 28 [Drawing Objects], page 254, on how to obtain the `XVisualInfo` for the window. Depending on the window manager, a colormap other than the default may not get installed correctly. If you're working with such a window manager, you may have to install the colormap yourself when the mouse pointer enters the canvas using `XInstallColormap()`.

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 (0) value for `yes_no`:

```
void fl_canvas_yield_to_shortcut(FL_OBJECT *obj, int yes_no);
```

To clear the canvas use

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

If `[fl_set_object_color()], page 287` gas been called on the object the first color passed to the function will be used to draw the background of the color, otherwise it's drawn in black.

21.3.5 Canvas Attributes

Some of the attributes, such as boxtype, do not apply to the canvas class.

The first color argument (`col1`) to `[fl_set_object_color()], page 287` can be used to set the background color of the canvas (by default, a canvas has no background color). The second argument (`col2`) controls the decoration color (if applicable).

21.3.6 OpenGL Canvas

Deriving specialized canvases from the general canvas object is possible. See the next subsection for general approaches how this is done. The following routines work for OpenGL (under X) as well as Mesa, 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 for a normal canvas. A "glcanvas" created this way 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()` for a list of valid attributes.

To get the current defaults use

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

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

```
void fl_set_glcanvas_attributes(FL_OBJECT *obj, const int *attributes);
```

Note that 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 *obj, int attributes[]);
```

The caller must supply the space for the attribute values.

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

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

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

```
void fl_set_glcanvas_direct(FL_OBJECT *obj, int yes_no);
```

with the argument `yes_no` set to false (0).

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

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

before drawing into glcanvas object.

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 *obj);
```

See demo program `gl.c` for an example use of a glcanvas.

22 Popups

Popup is not an object class. In contrast to normal objects popups are only shown for a short time in their own window and, while they are shown, no interaction with other objects is possible. So they don't fit directly into the normal event loop where one waits for user actions via [fl_do_forms()], page 296. Instead, when used stand-alone (e.g., for a context menu) they are shown on a call of the function [fl_popup_do()], page 211, which returns when the the user is done with the popup and it has been removed from the screen. Only idle callbacks and timers etc. are executed in the background while a popup is being shown.

Popups are the building blocks for menu and selector objects, which internally create and use popups. Thus it might be helpful to understand how popups work to get the most out of these objects.

All functions dealing with popups have names starting with 'fl_popup_', functions for individual entries start with 'fl_popup_entry_' and typedefs as well as macros with 'FL_POPUP_'.

22.1 Adding Popups

There are two ways to create and populate a popup with entries. The first method, that allows more fine-grained control consists of first generating a popup and then adding entries. Using this method all the properties of entries can be set immediately. The second method, to be discussed later, is simpler and may be sufficient for many applications, and internally uses the first method.

To define a new popup using the more general interface call

```
FL_POPUP *fl_popup_add(Window win, const char *title);
```

The function returns the address of the new popup on success and NULL on failure. win is the window of a parent object (use [FL_ObjWin()], page 198 to find out about it). You can also use [fl_root], page 301 for the root window, with None having the same effect . title is an optional string that gets shown at the top of the popup in a framed box. If not wanted pass an empty string or NULL. The function returns a pointer to a new popup or NULL on failure.

The title may contain embedded newline characters, this allows to create titles that span more than one line.)

There is no built-in limit to the number of popups that can be created.

Once you have popup you may add one or more entries by using

```
FL_POPUP_ENTRY *fl_popup_add_entries(FL_POPUP *popup,
                                     const char *entries, ...);
```

On success the return value is the address of the first entry created and NULL on failure. The first argument, entries, is a pointer to the popup the new entry (or entries) is added to. The second argument, entries, encodes information about the entries to add. In the most simple case it consists just of the entries texts, separated by | characters, e.g., "Item 1|Item 2|Item 3". This would create three simple entries in the popup with labels "Item 1", "Item 2" and "Item 3".

The `entries` string may contain newline characters which allows to create entries that span more than a single line.

There's no built-in limit to the number of entries than be added to a popup. [`fl_popup_add_entries()`], page 202 can be called repeatedly to append further entries to a popup.

It often is necessary to have more complex entries. E.g., one may want to have keyboard shortcuts for entries, which are shown on the right hand side of an entry, one may want to have sub-popups or set callbacks etc. This can be achieved by embedding special character sequences within the string describing the entries and passing further arguments to the function, similar to the use of a format string in e.g., `printf(3)`. All special sequences start with a `%`.

The following sequences are recognized:

`%x` Set a value of type `long int` that's passed to all callback routines for the entry. The value must be given in the arguments following the `entries` string.

`%u` Set a `user_void` pointer that's passed to all callbacks of the entry. The pointer must be specified in the arguments following the `entries` string.

`%f` Set a callback function that gets called when the entry is selected. The function is of type

```
int callback(FL_POPUP_RETURN *r);
```

Information about the entry etc. gets passed to the callback function via the `FL_POPUP_RETURN` structure (see below) and the return value of the function can be used to keep the selection being reported back to the caller of [`fl_popup_do()`], page 211 by returning a value of `FL_IGNORE` (-1). The functions address must be given in the arguments following the `entries` string.

`%E` Set a callback routine that gets called each time the mouse enters the entry (as long as the entry isn't disabled or hidden). The type of the function is the same as that of the callback function for the selection of the item but it's return value is never used. The functions address must be given in the arguments following the `entries` string.

`%L` Set a callback routine that gets called each time the mouse leaves the entry. The type of the function is the same as that of the callback function for the selection of the entry but it's return value is never used. The functions address must be given in the arguments following the `entries` string.

`%m` When this is specified a sub-popup gets opened when the mouse enters the entry (the entry itself thus can't be selected). The sub-popup to be opened must be an already existing popup and its address must be given in the arguments following the `entries` string. A triangle will be drawn on the right of the entry to indicate that it's an entry for a sub-popup.

Mutually exclusive with `%t`, `%T`, `%r`, `%R` and `%l`.

`%t`
`%T` This makes the entry a "toggle" entry, an entry that represents binary states and gets a check-mark drawn on its left if in "on" state. If created with `%t` its in "off" state at the start, if created with "T" its in "on" state. Switching states happens automatically when the entry is selected.

Mutually exclusive with `%m`, `%r`, `%R` and `%l`.

`%r`
`%R` This makes the entry a "radio" entry, i.e., it becomes part of a group of entries of which only one can be "on" at a time. The group, an integer value (don't use `INT_MIN` and `INT_MIN`), must be given in the arguments following the `entries` string.

Radio entries are drawn with a small circle to the left, with the one for the entry in "on" state filled with a color (blue per default). When a radio entry is selected by the user that was in "off" state the entry of the group that was is "on" state before is automatically switched to "off" state.

If the entry gets created with `%r` the entry is in "off" state, if created with `%R` it's in "on" state (in that case all entries created before in "on" state are reset to "off" state, i.e., the one created last "wins").

Mutually exclusive with `%m`, `%t`, `%T` and `%l`.

`%l` This creates not a real entry but indicates that a line is to be drawn to visually group other entries. While other properties can be set for such an "entry" only the "hidden" property (see below) is taken into acount.

Mutually exclusive with `%m`, `%t`, `%T`, `%` and `%R`.

`%d` Marks the entry as disabled, i.e., it can't be selected and its text is per default drawn in a different color

`%h` Marks the entry as hidden, i.e., it is not shown while in this state.

`%S` For entries with shortcut keys it's quite common to have them shown on the right hand side. Using `%S` you can split the entrys text into two parts, the first one (before `%S`) being drawn flushed left and the second part flushed right. Note that using this special sequence doesn't automatically sets a shortcut key, this still has to be done using `%s`.

`%s` Sets one or more shortcut keys for an entry. Requires a string with the shortcuts in the arguments following the `entries` string, see Section 26.1 [Shortcuts], page 245, for details on how to define shortcuts. Please note that the character in the label identical to the shortcut character is only shown as underlined if `%S` isn't used.

`%%` Use this to put a `%` character within the text of an entry.

Please note that since `[fl_popup_add_entries()], page 202` is a variadic function (i.e., it takes a variable number of arguments) only very limited error checking is possible and thus it is of importance that the arguments passed to the function have exactly the required types!

The return value of `[fl_popup_add_entries()], page 202` is a pointer to the first of the entries created. Since entries are stored as a linked list this value can be used to iterate over the list (see below for more information about the `[FL_POPUP_ENTRY], page 206` structure). If the function returns `NULL` no entries were created.

A typical piece of code creating a popup may look like this:

```
int save_cb(FL_POPUP_RETURN *result) {
```

```
    ...
}

int main(int argc, char *argv[]) {
   FL_POPUP *popup;
   File *fp;

   ...

   popup = fl_popup_add(None, NULL);
   fl_popup_add_entries(popup,
                        "Save%SCtrl+S%s%f%u|"
                        "Quit%SEsc%s|"
                        "%l|"
                        "Work Offline%SCtrl+O%T%s",
                        "^S", save_cb, (void *) fp,
                        "^[",
                        "^O");
    ...
}
```

This creates a popup with three entries. The first one has the label "Save" shown at the left and "Ctrl+S" at the right can be selected by pressing <Ctrl>S, in which case the function save_cb() will be invoked with a pointer to a structure that, beside other informations, contains the file pointer fp. The second entry has the labels "Quit" and "Esc" and it's shortcut key is set to <Esc>. Below this entry a separator line is drawn, followed by the third entry with labels "Work Offline" and "Ctrl+O" and shortcut key <Ctrl>O. This label is a "toggle" entry in "on" state, thus a check-marker is shown beside it.

A few remarks about the callback routines. All have a type of FL_POPUP_CB as given by this typedef:

```
typedef int (*FL_POPUP_CB)(FL_POPUP_RETURN *);
```

There are three kinds of callbacks, all with the same type. Whenever an item is entered (by moving the mouse on top of it or with the keyboard) its enter callback function is invoked (if one is set). Exceptions are entries that are disabled or hidden or entries, that just stand for separator lines. When an entry that can receive enter callbacks is left, its leave callback is invoked.

Leave callbacks are not called when a selection has been made. Instead, only the selection callback for the selected entry is invoked.

A "sub-popup entry", i.e., an entry that when entered results in a sub-popup to open, also can have an enter callback. Its leave callback is not called when the user moves the mouse onto the sub-popup but only once the sub-popup has been closed again and the mouse has been moved off the sub-popup entry.

While enter and leave callback functions are defined to return an integer value, it's never used. But for the third kind of callback, invoked on selection of an entry, this isn't true. Instead, the callbacks return value is important: if it is FL_IGNORE (-1), the selection isn't

reported back to the caller (and following callbacks also aren't called). This can be useful when the callback function already does everything required and nothing is left to be done.

All callbacks receive a pointer to a structure of the type FL_POPUP_RETURN:

```
typedef struct {
    long int              val;       /* value assigned to entry */
    void                 *user_data; /* pointer to user data */
    const char           *text;      /* text of selected popup entry */
    const char           *label;     /* text drawn on left */
    const char           *accel;     /* text drawn on right */
    const FL_POPUP_ENTRY *entry;     /* selected popup entry */
    const FL_POPUP       *popup;     /* (sub-) popup it belongs to */
} FL_POPUP_RETURN;
```

val is the value set by "%x". If "%x" wasn't given, it's an automatically generated value: when a popup is created with [fl_popup_add_entries()], page 202 a counter is initalized to 0. Whenever an entry gets added the value of the counter is assigned to the entry and then incremented. Unless a different value is set explicitely via "%x" the first entry added to a popup thus gets a value val of 0, the second one gets 1 etc. This even holds for entries that just stand for separator lines. In simple situations the value of val is probably sufficient to identify which entry got selected.

Please note: it is possible that by setting the val members two or more structures for items of the same popup get the same value. It is the programmers responsibility to avoid that (unless, of course, that's just what you intended).

The user_data member of the structure is the user_void pointer set via "%u". It allows to pass more complex data to the callback function (or have returned on selection of an entry.

The text member is exactly the string used to create the entry, including all the special sequences starting with '%'. label is what's left after all those sequences as well as backspace characters have been removed, tabs replaced by single spaces and the string is split at "%S". I.e., it's exactly what's drawn left-flushed for the entry in the popup. accel is then what's left after clean-up and came after "%S", i.e., it's what appears as the right-flushed text of the entry. Please note that one or more of these pointers could under some circumstances be NULL.

Finally, the two member entry and popup are pointers to the entry itself and the popup the callback function is invoked for - to find out the popup the selected entry itself belongs to use the popup member of the entrys [FL_POPUP_ENTRY], page 206 structure.

Please note: while in a callback you are only allowed to change the values of the val and user_data members. This can be useful in the case of a cascade of selection callback calls since all the selection callbacks receive the same structure (and this is also the structure that finally gets passed back to the caller of [fl_popup_do()], page 211) at the end in order to implement more complex information interchange between the callbacks involved.

The elements of a FL_POPUP_ENTRY structure that might be of interest) are

```
typedef {
    FL_POPUP_ENTRY *prev;    /* previous popup entry */
    FL_POPUP_ENTRY *next;    /* next popup entry */
    int             type;    /* normal, toggle, radio, sub-popup, line*/
    unsigned int    state;   /* disabled, hidden, checked */
```

```
        int             group;   /* group (for radio entries only) */
        FL_POPUP       *sub;     /* sub-popup bound to entry */
        ...
    } FL_POPUP_ENTRY;
```

Note that you should not change the members of a [FL_POPUP_ENTRY], page 206 structure directly! Use the appropriate functions documented below to modify them instead.

`prev` and `next` are pointers to the previous and the following popup entry (or `NULL` if none exists).

`type` tells what kind of popup entry this is. There are five different types:

`FL_POPUP_NORMAL`
: Normal popup entry with no special properties

`FL_POPUP_TOGGLE`
: "Toggle" or "binary" entry, drawn with a check-mark to its left if in "on" state

`FL_POPUP_RADIO`
: Radio entry, drawn with a circle to its left (color-filled when "on". The `group` member of the [FL_POPUP_ENTRY], page 206 structure determines to which group the entry belongs.

`FL_POPUP_SUB`
: Entry for a sub-popup. The `sub` member of its [FL_POPUP_ENTRY], page 206, structure is a pointer to the sub-popup that gets shown when the mouse enters the entry.

`FL_POPUP_LINE`
: Not a "real" entry, just indicates that a separator line is to be drawn between the previous and the next entry.

Finally, the `state` member can have the following values:

`FL_POPUP_NONE`
: No special state is set for the entry, the default.

`FL_POPUP_DISABLED`
: The entry is disabled, i.e., isn't selectable (and normally is drawn in a way to indicate this).

`FL_POPUP_HIDDEN`
: The entry is not drawn at all (and thus can't be selected).

`FL_POPUP_CHECKED`
: Only relevant for toggle and radio entries. Indicates that the state of a toggle entry is "on" (drawn with a check-marker) and for a radio entry that it is the one in "on" state of its group.

The state can be a combination of the above constants by using a bitwise OR.

The more interesting members of a `FL_POPUP` structure are

```
    typedef struct {
        FL_POPUP       *next;        /* previously created popup */
```

```
    FL_POPUP       *prev;        /* later created popup */
    FL_POPUP       *parent;      /* for sub-popups: direct parent */
    FL_POPUP       *top_parent;  /* and top-most parent */
    Window          win;         /* window of the popup */
    FL_POPUP_ENTRY *entries;     /* pointer to list of entries */
    char           *title;       /* title string of the popup */
    ...
} FL_POPUP;
```

Note again that you are not supposed to change the members of the structure.

Like popup entries also popups are stored in a (doubly) linked list. Thus the `prev` and `next` members of the structure are pointers to popups created earlier or later. If a popup is a sub-popup of another popup then `parent` points to the next higher level popup (otherwise it's `NULL`). In case there's a cascade of popups the `top_parent` member points to the "root" popup (i.e., the top-level popup), while for popups that aren't sub-popups it always points back to the popup itself (in that case `parent` is `NULL`).

`win` is the window created for the popup. It's `None` (0) while the popup isn't shown, so it can be used to check if the popup is currently visible.

The `entries` member points to the first element of the list of entries of the popup. See the `[FL_POPUP_ENTRY], page 206` structure documented above on how to iterate over all entries.

Finally, `title` is the title shown at the top of the popup (if one is set). Never try to change it directly, there ars the functions `[fl_popup_set_title()], page 215` and `[fl_popup_set_title_f()], page 215`, described below, to do just that.

To remove a popup entry use

```
int fl_popup_entry_delete(FL_POPUP_ENTRY *entry);
```

The function return 0 on success and -1 if it failed for some reasons. Note that the function for a sub-popup entry also deletes the popup that was associated with the entry!

You may also insert one or more entries into a popup at arbitrary places using

```
FL_POPUP_ENTRY *fl_popup_insert_entries(FL_POPUP *popup,
                                        FL_POPUP_ENTRY *after,
                                        const char *entries, ...);
```

`popup` is the popup the entries are to be inserted in, `after` is the entry after which the new entries are to be added (use `NULL` if the new entries are to be inserted at the very first position), and `entries` is the same kind if string as already used in `[fl_popup_add_entries()], page 202`, including all the available special sequences. The arguments indicated by `...` have to be given according to the `entries` string.

Finally, when you don't need a popup anymore simply call

```
int fl_popup_delete(FL_POPUP *popup);
```

The function returns 0 on success and -1 on failure. It's not possible to call the function while the popup is still visible on the screen. Calling it from any callback function is problematic unless you know for sure that the popup to be deleted (and sub-popups of it) won't be used later and thus normally should be avoided.

Above was described how to first generate a popup and then populate it. But there's also a (though less general) method to create and populate a popup in a single function call. For this use

```
FL_POPUP *fl_popup_create(Window win, const char *title,
                          FL_POPUP_ITEM *items);
```

The `win` and `title` arguments are the same as used in [fl_popup_add()], page 202, i.e., they are parent window for the popup (or [fl_root], page 301 or `None`) and the (optional, can be `NULL`) title for the popup.

`items` is a pointer to an array of structures of the following form:

```
typedef struct {
    const char  *text;        /* text of entry */
    FL_POPUP_CB  callback;    /* (selection) callback */
    const char  *shortcut;    /* keyboard shortcut description */
    int          type;        /* type of entry */
    int          state;       /* disabled, hidden, checked */
} FL_POPUP_ITEM;
```

The array must contain one structure for each entry of the popup and must end in a structure where at least the `text` member is set to `NULL`.

The `text` member describes the text of the entry. If it contains the string `"%S"` the text is split up at this position and the first part is used as the label drawn left-flushed for the entry and the second part for the right-flushed part (for showing accelerator keys etc.). Two more characters have a special meaning if they appear at the very start of the string (and which then do not become part of the label shown):

`'_'` Draw a separator line above this entry.

`'/'` This entry is a sub-popup entry and the following elements of the `items` array (until the first element with `text` set to `NULL` define the entries of the sub-popup.

Both `'_'` and `'/'` can appear at the start of the string, it doesn't matter which one comes first.

The `callback` member is a function to be invoked when the entry is selected (irrelevant for sub-popup entries). `shortcut` is a string, encoding which keyboard shortcut keys can be used to select the item (see Section 26.1 [Shortcuts], page 245, for details on how such a string has to be assembled).

`type` describes the type of the entry and must be one of [FL_POPUP_NORMAL], page 207, [FL_POPUP_RADIO], page 207 (all radio entries automatically belong to the same group (numbered `INT_MIN`). You can't use [FL_POPUP_LINE], page 207 or [FL_POPUP_SUB], page 207. If you want a sub-popup entry use [FL_POPUP_NORMAL], page 207 and set `'/'` as the first character of the `text` member of the structure. If you need a separator line put a `'_'` at the start of the `text` member string of the entry which comes after the separator line.

Finally, the `state` member can be 0 or the bitwise or of [FL_POPUP_DISABLED], page 207, [FL_POPUP_HIDDEN], page 207 and [FL_POPUP_CHECKED], page 207. The first one makes the entry appear disabled and non-selectable, the second will keep the entry from being drawn at all, and the third one puts the entry into `"on"` state (relevant for toggle and radio

entries only). If you try to set [FL_POPUP_CHECKED], page 207 for more than a single radio entry the last one you set if for "wins", i.e., only this one will be in "on" state. See below for a more detailed discussion of these entry properties.

[fl_popup_create()], page 209 does not allow to associate values or pointers to user data to individual entries, set titles for sub-popups, have radio entries belong to different groups or set enter or leave callback functions (though there exist a number of functions to remedy the situation in case such things are needed).

The function returns a pointer to the newly created popup (or NULL on failure). You are guaranteed that each entry has been assigned a unique value, starting at 0 and which is identical to the index of corresponding element in the items array, i.e., the first element results in an entry assigned 0, the second entry gets 1 etc.

All functions working on popups or entries can, of course, be used on popups and their entries generated via [fl_popup_create()], page 209. They can be employed to remedy some of the limitations imposed by the simpler popup creation API.

Here's an example of how to create a popup using fl_popup_create():

```
FL_POPUP *popup;

FL_POPUP_ITEMS items[] = {
  {"Item 1%S^1", NULL, "^1", FL_POPUP_NORMAL, FL_POPUP_NONE    },
  {"Item 2%S^2", NULL, "^2", FL_POPUP_RADIO,  FL_POPUP_CHECKED },
  {"Item 3%S^3", NULL, "^3", FL_POPUP_RADIO,  FL_POPUP_NONE    },
  {"_/Item 4",   NULL, NULL, FL_POPUP_NORMAL, FL_POPUP_NONE    },
    {"Sub-item A",  cbA, "^A", FL_POPUP_NORMAL, FL_POPUP_DISABLED},
    {"Sub-item B",  cbB, "^B", FL_POPUP_TOGGLE, FL_POPUP_NONE    },
    {NULL,         NULL, NULL, 0,               0                },
  {"Item 5",     NULL, NULL, FL_POPUP_NORMAL, FL_POPUP_NONE    },
  {NULL,         NULL, NULL, 0,               FL_POPUP_NONE    }
};

popup = fl_popup_create(None, "Test", items);
```

This creates a new popup with the title "Test" and 5 entries as well as a a sub-popup with two entries, that gets opened when the mouse is over the entry labeled "Item 4".

The first entry in the main popup has the label "Item 1" on the left and "^1" of the right side. It has no callback routine and can be selected via the <Crtl>1 shortcut. It's just a normal menu entry.

The second entry has the label "Item 2" on the left and "^2" of the right side, also no callack and <Crtl>2 as its keyboard shortcut. It's a radio entry that is in "on" state. The third entry is like the second, labels are "Item 3" and "^3" and it reacts to <Crtl>3, except that it's in "off" state. The second and third label belong to the same group (with the group number set to INT_MIN), i.e., when the third entry gets selected the second one gets switched to "off" state (and vice versa).

Before the fourth entry a separator line will be drawn (that's the effect of its text starting with '_'. It's a sub-popup entry (due to the '/' at the start of its text). It's label is simply "Item 4" and no right hand label (but that isn't supposed to indicate that sub-

entries couldn't have shortcuts!). It has no selection callback (which wouldn't sense make sense for a sub-popup entry anyway).

The following three elements of the `items` array are for the sub-popup that gets opened when the mouse is over the fourth item of the main popup. In the sub-popup we first have an normal entry with label `"Sub-item A"`. The function `cbA()` will be called when this entry of the sub-popup is selected. Then we have a second entry, labled `"Sub-item B"`, which is a currently disabled toggle entry in "off" state. If it weren't disabled its selection would result in the callback function `cbB()` getting called. The next element of the `items` array, having `NULL` as its `text` member, signifies the end of the sub-popup.

Now that we're done with the sub-popup another entry in the main popup follows, a normal entry with just a left-label of `Item 5`. The final element of `items`, where `text` is set to `NULL` then signifies that this is the end of the popup.

As there are functions to append to and insert entries into a popup with a kind of format string, followed by a variable list of arguments, there are also functions for adding and inserting entries using an array of `[FL_POPUP_ITEM], page 209`. These are

```
FL_POPUP_ENTRY *fl_popup_add_items(FL_POPUP *popup,
                                   FL_POPUP_ITEM *items);
FL_POPUP_ENTRY *fl_popup_insert_items(FL_POPUP *popup,
                                      FL_POPUP_ENTRY *after,
                                      FL_POPUP_ITEM  *items);
```

Both functions return the address of the first entry created on success and `NULL` on error. The first argument is the popup the entries are to be appended to or inserted into, the last argument the array of items (as in the case of `[fl_popup_create()], page 209` at least the `text` member of the last element must be a `NULL` pointer to indicate the end). `fl_popup_insert_items()` takes another argument, `after`, the entry after which the new entries are to be inserted (if called with `after` set to `NULL` the new entries are inserted at the very start of the popup).

22.2 Popup Interaction

A popup will be drawn on the screen when the function

```
FL_POPUP_RETURN *fl_popup_do(FL_POPUP *popup);
```

is called. It only returns when the user either selects an entry or closes it in some other way (e.g., by clicking outside the popup's area). When a selection was made the function returns a pointer to a `[FL_POPUP_RETURN], page 206` structure with information about the entry that was selected (please note that the structure is internal storage belonging to the Forms Library and is re-used when the popup is shown again, so copy out all data you may need to keep). If no selection was made (or one of the invoked callback routines returned a value of `FL_IGNORE` (-1) `NULL` is returned.

While the popup is shown the user can interact with the popup using the mouse or the keyboard. When the mouse is hovering over a selectable entry of the popup the entry is highlighted, when the mouse reaches an entry for a sub-popup, the associated sub-popup automatically gets opened. A selection is made by clicking on an entry (or, in case that the popup was opened while a mouse button was pressed down, when the mouse button is released). Clicking outside the popups window (or, depending on the "policy", see below,

releasing the mouse button somewhere else than over a selectable item) closes the popup without a selection being made.

Popups also can be controlled via the keyboard. First of all, on pressing a key, the shortcuts set for items are evaluated and, if a match is found, the corresponding entry is returned as selected (if the popup currently shown is a sub-popup, first the shortcuts for this sub-popup are checked, then those of its parent etc. until the top-most popup has been reached and checked for). The user can also navigate through the selectable entires using the `<Up>` and `<Down>` arrow keys and open and close sub-popups with the `<Right>` and `<Left>` cursor keys. Pressing the `<Home>` key highlights the first (selectable) entry in the popup, `<End>` the last one. By using the `<Esc>` key (or `<Cancel>` if available) the currently shown popup is closed (if an entry in a sub-popup was highlighted just this sub-popup is closed). Finally, pressing `<Return>` while on a selectable entry results in this entry being reported as selected.

Once the user has selected an entry its callback function is invoked with a `[FL_POPUP_RETURN], page 206` structure as the argument. When this function returns, the callback for the popup the entry belongs to is called with exactly the same structure. If the popup is a sub-popup, next the callback for its "parent" popup is invoked, again with the same structure (except that the `popup` member is changed each time to indicate which popup the call is made for). Repeat until the callback for the top-most popup has been called. Finally the structure used in all those callback invocations is returned from `[fl_popup_do()], page 211`. This chain of callback calls is interrupted when one of the callbacks returns a value of `FL_IGNORE` (-1). In that case no further callbacks are invoked and `[fl_popup_do()], page 211` returns `NULL`, i.e., from the callers perspective it looks as if no selection has been made. This can be useful when one of the callbacks was already was able to do all the work required on a selection.

Per default a popup stays open when the user releases the mouse button anywhere else than on a selectable entry. It only gets closed when the user either selects an entry or clicks somewhere outside of the popup area. An alternative is a "drag-down" popup that gets closed whenever the mouse button is released, even if the mouse isn't on the area of the popup or a selectable entry. To achieve this effect you can change the "policy" using the function

```
int fl_popup_set_policy(FL_POPUP *popup, int policy);
```

There are two values `policy` can have:

`FL_POPUP_NORMAL_SELECT`
: Default, popup stays open until mouse button is released on a selectable entry or button is clicked outside the popups area.

`FL_POPUP_DRAG_SELECT`
: Popup is closed when the mouse button is released anywhere.

The function can be called with either a (valid) popup address, in which case the policy for that popup is changed, or with a `NULL` pointer to change the default setting of the policy, used in the creation of new popups. The function returns the previous policy value or -1 on errors.

It's also possible to determine the policy setting by using

```
int fl_popup_get_policy(Fl_POPUP *popup);
```

If called with the address of a (valid) popup the policy for this popup (or its parent if one exists) gets returned. If called with a NULL pointer the default policy used in creating new popups is returned. On error -1gets returned.

Calling the function with NULL as the popup argument changes the default setting for the popups created afterwards.

If the popup is partially off-screen the user can push the mouse at the screen borders in the direction of the currently invisible popup entries. This results in the popups window getting moved so that previosuly invisible entries become accessible. The popup window gets shifted vertically in single entry steps, in horizontal direction by a tenth of the screen width. The delay between shifts is about 100 ms.

22.3 Other Popup Routines

When [fl_popup_do()], page 211 is called the popup per default is shown with its left upper corner at the mouse position (unless the popup wouldn't fit onto the screen). Using

```
void fl_popup_set_position(FL_POPUP *popup, int x, int y);
```

the position where the popup is drawn can be changed (but if it wouldn't fit onto the screen at that position it will also changed automatically). x and y. to be given relative to the root window, define the position of the upper left hand corner. Using this function for sub-popups is useless, they always get opened as near as possible to the corresponding sub-popup entry.

When setting the position of a popup it can be useful to know the exact sizes of its window in advance. These can be obtained by calling

```
int fl_popup_get_size(FL_POPUP *popup, unsigned int *w, unsigned int
*h);
```

The function returns 0 on success and -1 on error (in case the supplied popup argument isn't valid). Please note that the reported values are only valid until the popup is changed, e.g., by adding, deleting or changing entries or changing the appearance of the popup.

A callback function cb() of type [FL_POPUP_CB], page 205, to be called when a entry (or an entry of a sub-popup) is selected, can be associated with a popup (or changed) using

```
typedef int (*FL_POPUP_CB)(FL_POPUP_RETURN *);
FL_POPUP_CB fl_popup_set_callback(FL_POPUP *popup, FL_POPUP_CB cb);
```

The function returns the old setting of the callback routine (on error NULL is returned, which may indistinguishable from the case that no callback was set before).

For an entry all three associated callback functions can be set via

```
FL_POPUP_CB fl_popup_entry_set_callback(FL_POPUP_ENTRY *entry,
                                        FL_POPUP_CB cb);
FL_POPUP_CB fl_popup_entry_set_enter_callback(FL_POPUP_ENTRY *entry,
                                              FL_POPUP_CB enter_cb)
FL_POPUP_CB fl_popup_entry_set_leave_callback(FL_POPUP_ENTRY *entry,
                                              FL_POPUP_CB leave_cb);
```

The first function sets the callback invoked when the entry is selected, the second when the mouse enters the area of the entry and the third, when the mouse leaves that area. All function return the previously set callback or NULL when none was set or an error occured. NULL also gets returned on errors.

There are three properties that can be set for a popup entry

FL_POPUP_DISABLED
: The entry is is shown as disabled and can't be selected.

FL_POPUP_HIDDEN
: The entry is not shown when the popup is opened.

FL_POPUP_CHECKED
: Relevant only for toggle and redio entries. When set beside the label of a toggle entry a check-marker is drawn while the circle beside a radio button is drawn colored.

The "state" of an entry is the binary OR of these values which can be set and queried using the functions

```
unsigned int fl_popup_entry_set_state(FL_POPUP_ENTRY *entry,
                                      unsigned int state);
unsigned int fl_popup_entry_get_state(FL_POPUP_ENTRY *entry);
```

[fl_popup_entry_set_state()], page 214 returns the previous state on success and UINT_MAX (a value with all bits set) on failure.

Note that when setting [FL_POPUP_CHECKED], page 207 for a radio entry all other radio entries belonging to the same group automatically lose their "on" (checked) property.

There are also three convenience function for clearing, raising and toggling bits in the state of an entry. Normally to clear a certain bit of the state you have to first call [fl_popup_entry_get_state()], page 214, then clear the bit in the return value and finally call [fl_popup_entry_set_state()], page 214 with the result to set the new state. Use of these convenience functions allows to change state bits in a single call.

```
unsigned int fl_popup_entry_clear_state(FL_POPUP_ENTRY *entry,
                                        unsigned int what);
unsigned int fl_popup_entry_raise_state(FL_POPUP_ENTRY *entry,
                                        unsigned int what);
unsigned int fl_popup_entry_toggle_state(FL_POPUP_ENTRY *entry,
                                         unsigned int what);
```

The what argument can be any value resulting from a bitwise OR of [FL_POPUP_DISABLED], page 207, [FL_POPUP_HIDDEN], page 207 and [FL_POPUP_CHECKED], page 207 (thus you can clear, set or toggle one or more bits of the state in a single call). The functions all return the original value of the state.

You may search for an entry in a popup by different criteria (please note that the search also includes sub-popups of the popup, you can identify them by checking the popup member of the [FL_POPUP_ENTRY], page 206, structure). The search obviously will only deliver reasonable results if what you're searching for is unique between all the entries.

First, you can ask for the entry that had been created with a certain text, including all the special sequences, by calling

```
FL_POPUP_ENTRY *fl_popup_entry_get_by_text(FL_POPUP *popup,
                                           const char *text);
FL_POPUP_ENTRY *fl_popup_entry_get_by_text_f(FL_POPUP *popup,
                                             const char *fmt, ...);
```

The functions returns either a pointer to the entry found or `NULL` on failure (because either no entry with this text was found or the popup doesn't exist). (The functions differ in that the first one accepts just a simple string while the second assembles the text from a format string, just as it's used for `printf()` etc., and an appropriate number of following arguments.)

You may as well search by the left-flushed label parts of the entries as shown on the screen (note that tab characters `'\t'` originally embedded in the text used when creating the label have been replaced by single spaces and backspace characters `'\b'` were removed as well as all special sequences)

```
FL_POPUP_ENTRY *fl_popup_entry_get_by_label(FL_POPUP *popup,
                                            const char *label);
FL_POPUP_ENTRY *fl_popup_entry_get_by_label_f(FL_POPUP *popup,
                                              const char *fmt, ...);
```

Thus, since an entry created via a string like `"I\bt%Tem\t1%SCtrl+X"` will shown with a left-flushed label part of `"Item 1"`, this will be found when searching with either this string or a format string fo e.g., `"Item %d"` and a following integer argument of `1`.

Another way to search for an entry is by its value as either specified via the `"%x"` special sequence or assigned automatically by

```
FL_POPUP_ENTRY *fl_popup_entry_get_by_value(FL_POPUP *popup,
                                            long value);
```

Also the `user_data` pointer associated with the entry can be used as the search criterion:

```
FL_POPUP_ENTRY *fl_popup_entry_get_by_user_data(FL_POPUP *popup,
                                                void *user_data);
```

Finally one can try to find an entry by its current position in the popup (note that here sub-popups aren't taken into consideration since that would make the meaning of "position" rather hard to define) by

```
FL_POPUP_ENTRY *fl_popup_entry_get_by_position(FL_POPUP *popup,
                                               long position);
```

where `posistion` is starting with 0, so when called with 0 the first entry will be returned, when called with 1 you get the second entry etc. Note that separator lines aren't counted but entries currently being hidden are.

22.4 Popup Attributes

Using

```
void fl_popup_set_title(FL_POPUP *popup, const char *title);
void fl_popup_set_title_f(FL_POPUP *popup, const char *fmt, ...);
const char *fl_popup_set_title(FL_POPUP *popup);
```

the title of a popup can be changed or the currently set title determined. (The two functions for setting the title are just different in the way the title is passed: the first one receives a simple string while the second one assembles the title from a format string just like the one used with `printf()` etc. and an appropriate number of following arguments.)

To query or set the font the popups title is drawn in use

```
void fl_popup_get_title_font(FL_POPUP *popup, int *size, int *style);
```

```
void fl_popup_set_title_font(FL_POPUP *popup, int size, int style);
```

See Section 3.11.3 [Label Attributes and Fonts], page 27, for details about the sizes and styles that should be used. The default size and style are [FL_NORMAL_SIZE], page 27 and [FL_EMBOSSED_STYLE], page 28. This setting also applies to sub-popups of the popup, thus setting a title font for sub-popups is useless.

When called with the popup argument set to NULL the default settings for popups generated later are returned or set.

Also the font for the entries of a popup can be queried or and set via

```
void fl_popup_entry_get_font(FL_POPUP *popup, int *style, int *size);
void fl_popup_entry_set_font(FL_POPUP *popup, int style, int size);
```

The defalt size is [FL_NORMAL_SIZE], page 27 and the default style is [FL_NORMAL_STYLE], page 28. Again, the returned or set values also apply to all sub-popups, so calling the function for sub-popups doesn't make sense.

When called with popup set to NULL the default settings for popups are returned or changed.

The width of a popup is calculated using the widths of the title and the entries. You can influence this width by setting a minimum width a popup should have. There are two functions for the minimum width:

```
int fl_popup_get_min_width(FL_POPUP *popup);
int fl_popup_set_min_width(FL_POPUP *popup, int min_width);
```

The first one returns the currently set minimum width (a negative return value indicates an error). The second allows sets a new minimum width. Setting the minimum width to 0 or a negative value switches the use of the minimum width off. It returns the previous value (or a negative value on error).

You can query or set the border width popups are drawn width (per default it's set to 1). To this purpose call

```
int fl_popup_get_bw(FL_POPUP *popup);
int fl_popup_set_bw(FL_POPUP *popup, int bw);
```

Please note that the border width setting is automatically applied also to sub-popups, so there's no good reason to call these functions for sub-popups. The default border width is the same as that for objects.

The functions can also be called with popup set to NULL in which case the default setting for the border width is returned or set, respectively.

To change the cursor that is displayed when a popup is shown use

```
void fl_popup_set_cursor(FL_POPUP *popup, int cursor_name);
```

Use one of the symbolic cursor names (shapes) defined by standard X or the integer value returned by [fl_create_bitmap_cursor()], page 306 or one of the Forms Library's predefined symbolic names for the cursor_name argument.

Per default the cursor named "XC_sb_right_arrow" is used. If the function is called with popup set to NULL the default cursor for popups generated afterwards is changed.

There are several colors used in drawing a popup. These can be set or queried with the functions

```
FL_COLOR fl_popup_set_color(FL_POPUP *popup, int type,
                            FL_COLOR color);
```

```
FL_COLOR fl_popup_get_color(FL_POPUP *popup, int type);
```

where `type` can be one of the following values:

`FL_POPUP_BACKGROUND_COLOR`
: Background color of the popup, default is `FL_MCOL`.

`FL_POPUP_HIGHLIGHT_COLOR`
: Backgroud color an entry is drawn with when it's selectable and the mouse is on top of it, default is `FL_BOTTOM_BCOL`.

`FL_POPUP_TITLE_COLOR`
: Color used for the title text of a popup, default is `FL_BLACK`.

`FL_POPUP_TEXT_COLOR`
: Color normal used for entry texts, default is `FL_BLACK`.

`FL_POPUP_HIGHLIGHT_TEXT_COLOR`
: Color of the entry text when it's selectable and the mouse is on top of it, default is `FL_WHITE`.

`FL_POPUP_DISABLED_TEXT_COLOR`
: Color for drawing the text of disabled entries, default is `FL_INACTIVE_COL`.

`FL_POPUP_RADIO_COLOR`
: Color the circle drawn for radio entris in "on" state is drawn in.

When setting a new color the color previously used is returned by `[fl_popup_set_color()], page 216`. Calling these functions for sub-popups doesn't make sense since sub-popups are always drawn in the colors set for the parent popup.

When called with `popup` set to `NULL` the functions return or set the default colors of popups created afterwards.

To change the text of a popup entry call

```
int fl_popup_entry_set_text(FL_POPUP_ENTRY *entry, const char *text);
```

Please note that in the text no special sequences except `"%S"` (at which place the text is split to make up the left- and right-flushed part of the label drawn) are recognized.

The shortcut keys for a popup label can be changed using

```
void fl_popup_entry_set_shortcut(FL_POPUP_ENTRY *entry,
                                 const char *shortcuts);
```

See Section 26.1 [Shortcuts], page 245, for details on how such a string has to look like.

The value assigned to a popup entry can be changed via

```
long fl_popup_entry_set_value(FL_POPUP_ENTRY *entry, long value);
```

The function returns the previous value.

Also the user data pointer associated with a popup entry can be modified by calling

```
void *fl_popup_entry_set_user_data(FL_POPUP_ENTRY *entry,
                                   void *user_data);
```

The function returns the previous setting of `user_data`.

To determine to which group a radio entry belongs call

```
int fl_popup_entry_get_group(FL_POPUP_ENTRY *entry);
```

Obviously, this function only makes much sense when applied to radio entries. It returns the group number on success and `INT_MAX` on failure (that's why `INT_MAX` shouldn't be used for group numbers).

To assign a radio entry to a different group call

```
int fl_popup_entry_set_group(FL_POPUP_ENTRY *entry, int group);
```

Again, for obvious reasons, the function should normally only be called for radio entries. It returns the previous group number on success and `INT_MAX` on failure. If one of the entries of the new group was in `"on"` state the entries state will be reset to `"off"` if necessary.

For entries other than radio entries the group isn't used at all. So, theoretically, it could be used to store a bit of additional information. If that would be good programming practice is another question...

Finally, the sub-popup associated with a sub-popup-entry can be queried or changed using the functions

```
FL_POPUP *fl_popup_entry_get_subpopup(FL_POPUP_ENTRY *entry);
FL_POPUP *fl_popup_entry_get_subpopup(FL_POPUP_ENTRY *entry,
                                      FL_POPUP *subpopup);
```

Obviously, calling these functions only makes sense for sub-popup entries.

`[fl_popup_entry_get_subpopup()], page 218` returns the address of the sub-popup associated with the entry or `NULL` on failure.

To change the sub-popup of an entry a valid sub-popup must be passed to `[fl_popup_entry_set_subpopup()], page 218`, i.e., the sub-popup must not already be a sub-popup of another entry or the popup the entry belongs to itself. You also can't set a new sub-popup while the old sub-popup associated with the entry or the popup to become the new sub-popup is shown. On success the address of the new sub-popup is returned, on failure `NULL`.

Note that this function deletes the old sub-popup that was associated with the popup.

23 Deprecated Objects

In this chapter describes object types that have been replaced by newer ones. But they will remain part of XForms and also can be used in new programs. But there probably will be not more support for these objects than bug fixes etc.

23.1 Choice Object

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

23.1.1 Adding Choice Objects

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.

23.1.2 Choice Types

The following types are available:

`FL_NORMAL_CHOICE`
: Middle/right mouse button shortcut.

`FL_NORMAL_CHOICE2`
: Same as `FL_NORMAL_CHOICE` except drawn differently.

`FL_DROPLIST_CHOICE`
: Menu is activated only by pressing and releasing the mouse on the arrow.

23.1.3 Choice Interaction

Beside simply opening up the popup behind the choice object and selecting an entry with the left mouse button you can also use the middle and right mouse buttons and the scroll wheel: a short click with the middle mouse button selects the entry before the currently selected one, a click with the right mouse button the next. Keeping the middle or right mouse button pressed down slowly cycles trough the entries, backward or forward. The same can be down with the sroll wheel.

In both cases, whenever a choice entry is selected (even when it is the original one) the object is returned to the application program. But you can control the condition under which the choice object gets returned to the application by using the function

```
int fl_set_object_return(FL_OBJECT *obj, unsigned int when)
```

where `when` can have the following values

`[FL_RETURN_NONE], page 45`
: Never return or invoke callback.

`[FL_RETURN_END_CHANGED], page 45`
: Return or invoke callback if end of interaction and selection of an item coincide.

`[FL_RETURN_CHANGED], page 44`
: Return or invoke callback whenever an item is selected (this is the default).

`[FL_RETURN_END], page 44`
: Return or invoke callback on end of an interaction.

`[FL_RETURN_ALWAYS], page 45`
: Return (or invoke callback) whenever the interaction ends and/or an item is selected.

23.1.4 Other Choice Routines

There are a number of routines to change the list of possible choices. To add a line to a choice object use

```
int fl_addto_choice(FL_OBJECT *obj, const char *text);
int fl_addto_choice_f(FL_OBJECT *obj, const char *fmt, ...);
```

The function returns the number of the new item. The items in the list are numbered in the order in which they were inserted. The first item has number 1, etc. The two functions differ in that the first one accepts just a simple string while for the second the text is assembled from a format string as used by `printf()` etc. and the following arguments.

Note that, because a choice object uses a popup, the string passed with `[fl_addto_choice()], page 220` cann also contain some additional information not directly shown in the entries text. E.g., you can create several entries as once if the string you pass to `[fl_addto_choice()], page 220` contains `'|'` characters - these aren't shown but instead are treated as separators between the strings for the entries. Some extra control sequences, starting with the character `'%'` can also be embedded (see Section 23.3.1 [Creating XPopups], page 227), thus a literal `'%'` in a string must be escaped by doubling it.

To delete a line use:

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

Whenever the application program wants to clear the complete list of choices it should use the routine

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

One can also replace a line using

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

(The second function assembles the new text from a format string as used for `printf()` etc. and the following arguments.)

To obtain the currently selected item in the choice object use the call

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

The function returns the number of the current choice (0 if there is no choice).

You can also obtain the text of the currently selected choice item 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 an arbitrary choice item, use the following routine

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

To obtain the total number of choice 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 *obj, int numb, int mode);
```

Here `mode` is the same as that used for menu objects (see above). See also Section 23.3 [XPopup], page 227, for details.

To find about those settings use

```
int fl_get_choice_item_mode(FL_OBJECT *obj, int numb);
```

You can use the follow routine to populate a choice object at once, including mode and shortcut, by using

```
int fl_set_choice_entries(FL_OBJECT *obj, FL_PUP_ENTRY *entries);
```

where `entries` is a pointer to a `FL_PUP_ENTRY` structure (terminated by a `NULL` text field) as already described above for the function `[fl_set_menu_entries()], page 224`. Also see Section 23.3 [XPopup], page 227, for more details. Please note that for choice objects no nested entries are permitted and the item callback functions are ignored. The function returns the number of items added to the choice object.

Finally, the application program can set the currently selected entry of the choice using a call of

```
void fl_set_choice(FL_OBJECT *obj, int line);
void fl_set_choice_text(FL_OBJECT *obj, const char *txt)
void fl_set_choice_text_f(FL_OBJECT *obj, const char *fmt, ...)
```

where `txt` (for `fl_set_choice_text()` or the text resulting from the expansion of the `printf()`-compatible format string and the following arguments for `fl_set_choice_text_f()` must must be the text of exactly one of the choice items. For example, after the following choice is created

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

You can select the second item by using any of the following lines

```
fl_set_choice(obj, 2);
fl_set_choice_text(obj, "item2");
fl_set_choice_text_f(obj, "item%d", 2 );
```

23.1.5 Choice Attributes

Don't use `FL_NO_BOX` as the boxtype for a choice object.

The first color argument (`col1` to `[fl_set_object_color()], page 287` controls the color of the box and the second (`col2`) 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 *obj, int align);
```

To set the font size used inside the choice object use

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

To set the font style used inside the choice object use

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

Note that the above functions only change the font inside the choice object, not the font used in the popup. To change the font used in the popup, use the XPopup functions [fl_setpup_default_fontsize()], page 235 and [fl_setpup_default_fontstyle()], page 235. Note that these functions influence the font settings of all popups! See Section 3.11.3 [Label Attributes and Fonts], page 27, for details on font sizes and styles.

Normally the pop-up shown for the choice objects will be displayed at the current mouse position or, for those of type FL_DROPLIST_CHOICE, directly below the choice object. This can be modified by a call of the function

```
int fl_set_choice_align_bottom(GL_OBJECT *obj, int flag);
```

If flag is 0 the normal behaviour is used, but when flag is 1 the popup will be displayed with its lower right hand corner at the current mouse position or, for objects of type FL_DROPLIST_CHOICE above the choise object. The function returns the previously set value for flag.

23.1.6 Remarks

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

23.2 Menu Object

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.

23.2.1 Adding Menu Objects

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.

23.2.2 Menu 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.

FL_PUSH_MENU and FL_PULLDOWN_MENU behave rather similar. When you click on a FL_PUSH_MENU per default a pop-up window gets opened on top of the FL_PUSH_MENU menu's

box that has a label at the top, indicating the currently selected menu item. The pop-up window stays open until you either select an item or press a mouse button somewhere outside the pop-up window.

When you click on FL_PULLDOWN_MENU also a pop-up window is shown, but directly below the menu's box. This pop-up window has no label and it only stays open until you release the mouse button.

FL_PUSH_MENU and FL_PULLDOWN_MENU can be made even more similar by using the [fl_set_menu_notitle()], page 227 function (see below). This changes it's properties so that the pop-up window also appears below the menu's box and that no label is shown in the pop-up window. The only remaining difference then is that a FL_PUSH_MENU only gets closed when a menu item is selected or the user presses the mouse outside of the pop-up window while a FL_PULLDOWN_MENU also gets closed when the mouse button is released.

23.2.3 Menu Interaction

When the menu appears the user can make a selection using the left mouse button or make no selection by clicking outside the menu (or by simply releasing the mouse button in case of a FL_PULLDOWN_MENU type menu. Normally when he makes a selection the menu object is returned by the interaction routines.

You can control the condition under which the menu object gets returned to the application by using the function

```
int fl_set_object_return(FL_OBJECT *obj, unsigned int when)
```

where when can have the following values

[FL_RETURN_NONE], page 45
: Never return the object or invoke its callback.

[FL_RETURN_END_CHANGED], page 45
: Return or invoke callback if end of interaction and selection of an item coincide (this is the default for all menu objects except those of type FL_TOUCH_MENU).

[FL_RETURN_CHANGED], page 44
: Return or invoke callback whenever an item is selected (this is the default for all menu objects of type FL_TOUCH_MENU).

[FL_RETURN_END], page 44
: Return or invoke callback on end of an interaction.

[FL_RETURN_ALWAYS], page 45
: Return (or invoke callback) whenever the interaction ends and/or an item is selected.

23.2.4 Other Menu Routines

There are two ways to populate a menu, i.e., add items. The first one is a bit more complex but allows for more flexibility, e.g., later adding and removing items, associating callbacks with individual items etc. For the more simple (and in many cases sufficient) method see the function [fl_set_menu_entries()], page 224.

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 23.3 [XPopup], page 227). In the simplest case it just contains the texts for the menu items, separated by a bar (`'|'`), e.g., `"First|Second|Third"`. But it's also possible to employ special tags (see Section 23.3.1 [Creating XPopups], page 227) that can be used to indicate special attributes (radio, toggle and greyed-out, for example). For this usage the unspecified arguments (the `...` in the function call) can be used t add neceassary information. Whenever the user selects a menu item, a pointer to the menu object it belongs to is returned to the application program.

Please note that if you call `[fl_set_menu()], page 223` on a menu that already contains items those items are removed. The function calls `[fl_clear_menu()], page 225` internally before the new items are added.

If you explicitely assign a menu item ID to a menu, using the special tag `%x`, it is your responsibility to make sure that this ID isn't already used by a different menu item in the same menu. Failure to do so may make it impossible to use the menu properly. All functions working on items expect the menu item ID as one of their arguments.

In case you don't set menu item IDs they are assigned automatically with the first item obtaining the menu item ID 1, the next 2 etc., i.e., it directly reflects the position of the item in the menu.

It is also possible to add menu items to an existing menu using a call of

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

where `menustr` is a string of the same form as used in `[fl_set_menu()], page 223` (you can add one or more new menu items this way).

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

```
void fl_delete_menu_item(FL_OBJECT *obj, int miid);
void fl_replace_menu_item(FL_OBJECT *obj, int miid,
                          const char *menustr, ...);
```

`miid` is the menu item ID. `menustr` must be a string as used in `[fl_set_menu()], page 223` with the only difference that only a single menu item can be specified.

Please note: when deleting a menu item all other items keep their menu item IDs. The menu item ID of the deleted menu item isn't re-used when new items are added later. Instead for each menu an internal counter exists that gets incremented for each menu item added and which value is used for the menu item ID unless one is explicitely assigned to the menu item. The counter oly gets reset to 1 when the menu is cleared used `[fl_clear_menu()], page 225`.

The menu item ID of a menu item changed by using `[fl_replace_menu_item()], page 224` does not change unless the library is explicitely asked to via `%x` in `menustr`.

For most applications, the following routine may be easier to use at the expense of somewhat restrictive value a menu item can have as well as a loss of the ability to delete menu items or associate callbacks with menu items.

```
int fl_set_menu_entries(FL_OBJECT *obj, FL_PUP_ENTRY *ent);
```

where `ent` is a pointer to an array of structure of the following type, terminated by an element, where at least the `text` member is a `NULL` pointer:

```
typedef struct {
```

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

The meaning of each member is explained in Section 21.3. For menus, item callback function can be `NULL` if the menu callback handles the interaction results. See demo program `popup.c` for an example use of `[fl_set_menu_entries()], page 224`.

The function `[fl_set_menu_entries()], page 224` works by creating and associating a popup menu with the menu object. The popup ID is returned by the function. Whenever the function is called, the old popup associated with the object (if one exists) is freed and a new one is created. Although you can manipulate the menu either through the menu API (but adding and removing menu items is not supported for menus created this way) or popup API, the application should not free the popup directly and use `[fl_clear_menu()], page 225` instead.

To clear the whole menu use

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

To find the menu item selected by the user use

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

The the function returns the menu item ID. In the simplest possible case this is just the position of the menu item (starting at 1). This stops to be true when either IDs have been explicitely assigned to items or items have been deleted. In that case the following rules apply:

1. A menu item ID may have been assigned to a menu item using `%xn` in the string for the text of the menu item.
2. Menu items can get associated with a callback function that is executed when the menu item is selected. The callback function is of type `[FL_PUP_CB], page 233` and receives the menu item ID of the selected menu. If such a callback is set for a menu item the return value of `[fl_get_menu()], page 225` is the return value of this function instead of the menu item ID that would have been returned otherwise.

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

```
const char *fl_get_menu_item_text(FL_OBJECT *obj, int miid);
```

where `miid` is the menu item ID. If `n` isn't a valid menu iem ID item `NULL` is returned.

To obtain the text of the selected enu item use

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

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

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

One can change the appearance of different menu items. In particular, it is sometimes desirable to make grey-out menu items and make them unselectable or 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 miid, unsigned mode);
```

`miid` is the menu index ID of the memu item you want to change. `mode` represents the special properties you want to apply to the chosen item. 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_BOX`
: "Binary" entry, i.e., an entry that stands for a choice that can be switched on and off. Displayed with an unchecked box to the left.

`FL_PUP_RADIO`
: "Radio" item belonging to a group, so that gets automatically switched off when another item of the group is selected. Displayed with a diamoned-shaped box at the left.

`FL_PUP_GREY`
: To be OR-ed with one of the above to make that item appear greyed-out and disable it (i.e., not selectable anymore).

`FL_PUP_CHECK`
: To be OR-ed with one of `FL_PUP_BOX` and `FL_PUP_RADIO` to make the box to the left appear checked or pushed.

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 *obj, int miid);
```

While a callback associated with a menu entry can be set when it is created it can also set later on or be changed. For this use the function

```
FL_PUP_CB fl_set_menu_item_callback(FL_OBJECT *ob,
                                    int numb, FL_PUP_CB cb);
```

where `numb` is the menu entries ID and `cb` is the callback function of type `[FL_PUP_CB], page 233` (or `NULL` to disable a callback). The return value is a pointer to the previously used callback function (or `NULL`).

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 miid,
                               const char *str);
```

`miid` is the menu item ID of the menu item under consideration. `str` contains the shortcut for the item. (Actually, it can contain more shortcuts for the same item.) See Section 26.1 [Shortcuts], page 245, for more information about shortcuts.

Finally there is the routine:

```
void fl_show_menu_symbol(FL_OBJECT *obj, int yes_no);
```

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.

23.2.5 Menu Attributes

Any boxtype can be used for a menu except for those of type `FL_PULLDOWN_MENU`, for which `FL_NO_BOX` should not be used.

Using the functiond

The first color argument (`col1`) to `[fl_set_object_color()], page 287` controls the color of the menu's box when not open and the second (`col2`) is the color when the menu is shown.

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

```
void fl_setpup_default_fontstyle(int style);
void fl_setpup_default_fontsize(int size);
```

These settings apply to all menus at once.

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

```
void fl_set_menu_popup(FL_OBJECT *obj, int pupID);
```

where `pupID` is the ID returned by `[fl_newpup()], page 227` or `[fl_defpup()], page 227`. See Section 23.3 [XPopup], page 227, for more details on popup creation.

For a menu created this way only `[fl_get_menu()], page 225` and `[fl_get_menu_text()], page 225` work as expected. Other services such as mode setting and query etc. should be done via the popup routines.

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

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

The function returns the popup ID if the menu was created using `[fl_set_menu_popup()], page 227` or `[fl_set_menu_entries()], page 224`, otherwise it returns -1.

The callback associated with a menu

Normally in the popup opened for a menu a title is shown. This can be switched off (and back on again by using the function

```
fl_set_menu_notitle(FL_OBJECT *obj, int off);
```

23.2.6 Remarks

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

23.3 XPopup

XPopup is not really an object class, but because it is used by menu and choice objects and can function stand-alone, it is documented here.

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

23.3.1 Creating XPopups

To define a new popup, use the following routines

```
int fl_newpup(Window parent);
int fl_defpup(Window parent, const char *str, ...);
```

Both functions allocate and initialize a new popup menu and return the XPopup identifier (or -1 on failure). [fl_defpup()], page 227 in addition accepts a pointer `str` to the texts for menu items (optionally also some more arguments, see below). 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 XPopup belongs. In a situation where the XPopup is used inside an object callback `FL_ObjWin(obj)` will do. If `parent` is `None` the root window will be used.

Calling [fl_defpup()], page 227 with the `str` argument set to `NULL` is equivalent to calling [fl_newpup()], page 227.

It is possible to specify XPopup and item properties, such as shortcuts, callbacks etc., together with the items texts using a format string system similar as used for e.g., `oprint(3)`. If XPopup or item properties require arguments, they must be passed to [fl_defpup()], page 227 following the `str` argument.

The following item properties are supported:

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

`%F` Binds a callback function to the XPopup as a whole that is called for every selection made from this XPopup. You must specify the function to be invoked in the parameters following `str`. The value of the selected item is passed as the only argument to the invoked callback function. The callback function must return a non-negative integer. If such a callback function has been registered for a XPopup and you select its third item, in the simplest case 3 will be passed as a parameter to the callback function (more complicated situations would involve that the item had been assigned a different value. e.g., using `%x`, see below, or that there's also a callback bound to the item itself, in which case the global XPopup callback would receive the return value of the items callback function).

`%f` Binds a callback to this particular item which is invoked if the item is selected. The routine must be supplied in the parameters following `str`. It has to return a non-negative integer. The value of the selected item is passed as a parameter to this function. If you have also bound the entire XPopup to a callback function via `%F`, then the function specified via `%f` is called first with the items value and its return value (if larger then `0` is then passed as the parameter to to the function bound to the whole XPopup (as set via `%F`).

`%i` Disables and greys-out the item. `%d` can be used instead of `%i`.

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

`%m` Whenever this item is selected another (already defined) XPopup is bound to the item so that the sub-XPopup is opened when the user moves the mouse onto the item, This can be used to create cascading menus. The identifier of the sub-XPopup to be shown must be provided in the arguments following `str`. It is the programmers responsibility to make sure that the item values of the sub-XPopup don't clash with those of the higher-level XPopup or it may be impossible to determine which item was selected.

%h
: Specify a "hotkeys" that can be used to select this item. Hotkeys must be given in the arguments following str as a pointer to a string. Use # to specify that a key must be pressed together with the <Alt> key, ^ for simultaneous pressing of <Ctrl> and &n for the function key Fn.

  %s can be used instead of %h.

%xn
: Assigns a numerical value to this item. This value must be positive. This new value overrides the default position-based value assigned to this item. Different from most other flags, the value n must be entered as part of the text string (i.e., do not try to use the arguments following str to specify this value!) and must be number larger than 0. It is the programmers responsibility to make sure that the items value does not clash with those of other items of the XPopup or determining which item was selected may be impossible.

%b
: Indicates this item is "binary item" (toggle), currently in off state. When displayed, binary items will be drawn with a small box to the left. See also FL_PUP_BOX.

%B
: Same as %b except that it also signifies that this item is in on or "true" state and consequently is drawn with a checked box on the left. See also FL_PUP_BOX | FL_PUP_CHECK.

%rg
: Specifies this menu item is a "radio item" belonging to group with number g, currently not being selected. The group number g, that must be part of the string directly following %r (and not specified via the arguments following the string), must be a non-zero, positive number. Radio items are drawn with a small diamond box to the left (empty while not active). See also FL_PUP_RADIO.

%Rg
: Same as %rg except that it also sets the state of the radio item as selected or "pushed", the item is drawn with a filled diamond box to the left. See also [fl_setpup_selection()], page 235. See also FL_PUP_RADIO | FL_PUP_CHECK.

%%
: Use this if you need a % character in the string.

<Ctrl>H (\010)
: Same as %l except that the character must precede the item label, i.e., use "\010Abc" and not "Abc\010".

Due to the use of variable arguments error checking can only be minimal. Also note that if %x is used to specify a value that happens to be identical to a position-based value, the result is unpredictable when subsequent references to these items are made. There is currently a limit of FL_MAXPUPI (64) items per popup.

Tabs characters ('\t') can be embedded in the item string to align different fields.

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

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

where popup_id is the value returned by [fl_newpup()], page 227 or [fl_defpup()], page 227 for the XPopup. Again, str can contain information for one or more new items, including the special sequences described earlier. The function returns -1 if invalid arguments are detected (as far as possible for a function with a variable number of arguments).

To display a popup, use

```
int fl_dopup(int popup_id);
```

This function displays the specified XPopup until the user makes a selection or clicks somewhere outside of the XPopups box. The value returned is the value of the item selected or -1 if no item (or a disabled one) was selected. However, if there is a function bound to the XPopup as a whole or to the selected item itself, this function is invoked with the item value as the argument and the value returned by [fl_dopup()], page 230 is then the return value of this function. If a callback function for both the selected item and the XPopup as a whole exists, the callback function for the item is called first with the item value as the argument and then the return value of this item specific callback function is passed to the XPopups callback function. [fl_dopup()], page 230 then finally returns the return value of this second function call.

Normally a XPopup get opened when the left mouse button has been pressed down and get closed again when the left mouse button is released. But there are a number of ways to achieve a "hanging" XPopup, i.e., that the XPopup that says open, even though the left mouse button isn't pressed down anymore. This happens e.g., when the user releases the mouse button in the title area of the XPopup or when the XPopup was opened via a keyboard shortcut. In that case it's also possible to navigate through the items and select via the keyboard.

A typical procedure may look as follows:

```
int item3_cb(int n) {
     return n + 7;
}

/* define the menu */
int menu = fl_newpup(parent);
fl_addtopup(menu, "Title %t|Item1%rg1|Item2%Rg1|Item3%x10%f|Item4",
            item3_cb);

switch (fl_dopup(menu)) {
    case 1:   /* item1 is selected */
        /* handle it */
        break;

    case 2:
        /* handle it */
        break;

    case 4:
        /* handle it */

    case 17:
        /* item 3 call back has been executed */
}
```

Here callback function `item3_cb()` is bound to the third item and this item has been assigned the number 10. Thus, when it is selected [fl_dopup()], page 230 does not return

3 or 10. Instead the callback function `item3_cb()` is invoked with 10 as its argument. And this function in turn returns `10 + 7`, which is the value `[fl_dopup()], page 230` finally returns.

Note also that items 1 and 2 both are radio items, belonging to the same group (numbered 1). Item 2 is currently the active item of this group.

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

```
int fl_current_pup(void);
```

If no popup is active, the function returns -1. Until all callback functions have been run the function returns the ID of the XPopup the items belong to.

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

```
void fl_freepup(int popup_id);
```

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

```
void fl_setpup_entries(int popup_id, FL_PUP_ENTRIES *entries);
```

where `popup_id` is the popup ID returned by `[fl_newpup()], page 227` or `[fl_defpup()], page 227` and `entries` is an array of the following structures

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

The meaning of each member of the structure is as follows:

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

- `'/'` This indicates the beginning of a sub-popup, starting with the next item and ending with the next item with `text` being `NULL`.
- `'_'` Indicates that a line should be drawn below this item (typically as a visual reminder of logical groupings of items).

`callback` This is the callback function that will be called when this particular item is selected by the user. `[fl_dopup()], page 230` returns the value returned by this callback. If the callback is `NULL`, the item number will be returned directly by `[fl_dopup()], page 230`.

`shortcut` Specifies the keyboard shortcut.

`mode` Specifies special attributes of this item. This can be one or a combination by bitwise OR of one of the following:

`FL PUP NONE`
: No special characteristics, the default.

`FL_PUP_GREY`
: Item is greyed-out an can't be selected. Trying to select it results in `[fl_dopup()], page 230` returning -1.

`FL_PUP_BOX`
: "Binary item", drawn with a little box to its left.

`FL_PUP_RADIO`
: "Radio item", drawn with a little diamond-shaped box to its left. All radio items of the XPopup belong to the same group.

`FL_PUP_CHECK`
: OR this value with `FL_PUP_BOX` or `FL_PUP_RADIO` to have the box to the left drawn as checked or pushed.

With this simplified API, popup item values start from 1 and are the index in the entries array for the item plus 1. For example, the third element (with index 2) of the array of structure has an item value of 3. Please note that also elements of the array that end a submenu and thus don't appear as visible items in the XPopup get counted. This way, the application can relate the value returned by fl_dopup() to the array easily. See demo program `popup.c` for an example use of the API.

To illustrate the usage of `[fl_setpup_entries()], page 231`, Fig 21.2 shows the popup created with the array of structures defined in the following code example:

```
FL_PUP_ENTRY entries[ ] = {
   {"Top item1",  callback},      /* item number 1 */
   {"Top item2",  callback},
   {"Top item3",  callback},
   {"/Top item4", callback},
     {"Sub1 item1",  callback},   /* item number 5 */
     {"Sub1 item2",  callback},
     {"Sub1 item3",  callback},
     {"Sub1 item4",  callback},
     {"/Sub1 item5", callback},
       {"Sub2 item1",  callback}, /* item number 10 */
       {"Sub2 item2",  callback},
       {"Sub2 item3",  callback},
       {NULL,         NULL     }, /* end of level2, item number 13 */
     {NULL,           NULL   },   /* end of sublevel1, item nuber 14 */
   {"Top item5",  callback},      /* item number 15 */
   {NULL,         NULL    }       /* end of popup */
};
```

23.3.2 XPopup Interaction

To select an item, move the mouse to the item to be selected while keeping the mouse button pressed down and then release the mouse button on top of the item to be selected. If you don't want to make a selection release the mouse button somewhere outside the area of the XPopup.

If you have a "hanging" XPopup, i.e., a XPopup that's open even though the mouse button isn't pressed anymore you can select by clicking on an item or use the cursor `Up` and `Down` keys to navigate through the items and select by pressing the `<Return>` key. The `<Home>` and `<End>` keys allow you to jump to the first or last selectable item, respectively. Use `<Esc>` to close the popup without selecting an item.

It is also possible to use convenience functions to bind keyboard keys to items (the "hotkeys") instead of using %s with [fl_defpup()], page 227:

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

where item_val is the value associated with the item (either due to its position or set with %x) and hotkeys is a string specifying all the hotkey combinations. See Section 26.1 [Shortcuts], page 245, for details. Briefly, within that string # and ^ denote the <Alt> and <Ctrl> keys, respectively. &n with n = 1, 2 etc. can be used to denote the function key numbered n. Thus if hotkeys is set to "#a^A, both <Ctrl>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 key in the hotkey string is used. Therefore, the hotkey strings "Cc", "#C" and "^C" will result in the character C in the item string "A Choice" being underlined, while the the hotkey strings "cC" and "#c" will not since there's no c in the item string. There is a limit of maximum 8 shortcut keys.

Two convenience functions are available to set the callback functions for items of a XPopup and the XPopup as a whole (called whenever a selection is made):

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

These functions thus allow to change the popup and item callback functions set at creation of the popup with %F and %f. As usual, popup_id is the ID of the XPopup, item_val the value associated with the item (position or value set via %x), and cb is the address of the callback function.

Please note that Xpopup objects are a bit special in XForms. Normal objects get returned by e.g., [fl_do_forms()], page 296 (or an associated callback gets invoked). But since Xpopup objects are meant to be sub-objects of other objects (like FL_CHOICE and L_MENU objects) and don't get invoked directly by a call of e.g., [fl_do_forms()], page 296 but instead by a call of [fl_dopup()], page 230 they can't get returned to the application. Instead the caller of [fl_dopup()], page 230 (normally some internal function of a FL_CHOICE or FL_MENU object) has to deal with the return value.

Furthermore, also callback functions can be set that get invoked whenever an item in the XPopup is entered or left, even without a selection being made. The following functions can be used to register these item enter/leave callbacks:

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

FL_PUP_ENTERCB fl_setpup_entercb(int popup_id,
                                 FL_PUP_ENTERCB cb, void *data);
FL_PUP_LEAVECB fl_setpup_leavecb(int popup_id,
                                 FL_PUP_LEAVECB cb, void *data);
```

The function cb will be called when the mouse enters or leaves an (non-disabled) item of the XPopup popup_id. Two parameters are passed to the callback function. The first parameter is the item number enter/leave applies to and the second parameter is a data pointer. To remove an enter/leave callback, call the functions with the callback function argument cb set to NULL.

There is also a function to associate a XPopup item with a sub-XPopup

```
void fl_setpup_submenu(int popup_id, int item_val, int subpopup_id);
```

If a sub-XPopup is associated with item `item_val` that item can't be selected anymore (releasing the mouse button on this item makes `[fl_dopup()], page 230` return -1 but instead a new XPopup is opened beside the item and you can now make selections within this sub-XPopup. It is the programmers responsibility to make sure that the item values of the sub-XPopup don't clash with those of the higher-level XPopup or it may be impossible to determine which item was selected.

23.3.3 Other XPopup Routines

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

It is possible to modify the display characteristics of a given XPopup item after its creation using the following routine

```
void fl_setpup_mode(int popup_id, int item_val, unsigned mode);
```

As usual `popup_id` is the XPopup ID as returned by `[fl_newpup()], page 227` or `[fl_defpup()], page 227` and `item_val` the value of the item. `mode` is one of `FL PUP NONE`, `FL PUP GREY`, `FL PUP BOX` or `FL PUP RADIO` (one of the later two can be bitwise ORed with `FL_PUP_CHECK`, as already discussed above.

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

```
unsigned int fl_getpup_mode(int popup_id, int item_val)
```

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

```
if (fl_getpup_mode(popupd, item_val) & FL_PUP_CHECK)
    /* item is set */
```

There exists also a routine that can be used to obtain an items text

```
const char *fl_getpup_text(int popup_id, int item_val);
```

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

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

where `x` and `y` specify the location where the top-left corner of the popup should be. `x` and `y` must 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()], page 230`, the position is not remembered afterwards.

If `x` or `y` is negative, the absolute value is taken to mean the desired location relative to the right or bottom corner of the popup (not the screen!).

Another function exists for controlling the positon of the popup. When the fuunction

```
void fl_setpup_align_bottom(void);
```

then the pop-up will appear with its lower right hand corner aligned aligned with the mouse position or, if also `[fl_setpup_position()], page 234` is active, the postion set this way will be interpreted to mean the lower right hand position of the popu-up.

A radio item in a group can be initialized to be in "pushed" state by using `%R`. But you can also switch a such a radio item to "pushed state also programmatically using

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

Of course, other radio items of the XPopup belonging to the same group are reset to "unpushed" state.

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

```
int fl_getpup_items(int popup_id)
```

23.3.4 XPopup Attributes

The title of a XPopup can be set using the functions

```
void fl_setpup_title(int popup_id, const char *title);
void fl_setpup_title_f(int popup_id, const char *fmt, ...);
```

They only differ in the way the new title is passed to the function, the first one accepts a simple string while the second expects a format string as used for `printf()` etc., followed by the appropriate number of (unspecified) arguments.

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

```
int fl_setpup_default_fontsize(int size);
int fl_setpup_default_fontstyle(int style);
int fl_setpup_default_bw(int bw);
```

The functions return the old size, style or border width value, respectively.

All XPopups by default use a right arrow cursor. To change the default cursor, use

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

where you can use for `cursor` any of the standard cursors defined in `<X11/cursorfont.h>` like `XC_watch` etc. The function returns the previously cursor.

To change the cursor of a particular XPopup only , use the following routine

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

For example, after the following sequence,

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

the popup with ID `id` will use a "hand" instead of the default arrow cursor.

In versions before 1.0.91 XPopups were drawn with a heavy shadow around the box. Drawing of this shadow could be controlled via

```
void fl_setpup_shadow(int popup_id, int yes_no);
```

Nowadays this function still exists for backward-compatibility but does nothing.

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

```
void fl_setpup_bw(int popup_id, int bw);
void fl_setpup_softedge(int pupup_id, int yes_no);
```

The first sets the border width for a XPopup. Calling [`fl_setpup_softedge()`], page 235 with a true argument for `yes_no` has the same effect as using a negative border width while

using a false (0) argument is equivalent to using a positive one (so this function isn't very useful).

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

```
void fl_setpup_default_color(FL_COLOR bgcolor, FL_COLOR tcolor);
```

By default, the background color `bgcolor` is `FL_COL1` and the text color `tcolor` is `FL_BLACK`.

For "binary" or radio items, that have check box associated with them, the "checked" or "pushed" color (default is `FL_BLUE`) can be changed with the following routine

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

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

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

The function returns the previous limit.

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

```
void fl_showpup(int popup_id);
void fl_hidepup(int popup_id);
```

No interaction takes place with a XPopup shown by `[fl_showpup()], page 236` and it can only be removed from the screen programmatically via `[fl_hidepup()], page 236`.

23.3.5 Remarks

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

XPopups are used indirectly in the demo programs `menu.c`, `boxtype.c`, `choice.c` and others. For a direct pop-up demo see `popup.c`.

Part IV - Designing Object Classes

24 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 can be situations where it is beneficial to create new types of objects, for example switches or joysticks or other types of sliders, etc. In these cases, a programmer can use the architecture defined by the Forms Library to create a 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 a free object. This part gives you all the details of how to add new classes. In chapter 24 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 25 describes in detail what type of events objects can receive and how they should react to them. Chapter 26 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 27 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 28 gives an example illustrating on 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 29 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 30.1 discusses the possibilities of using the pre-emptive handler of an object to implement derived objects.

25 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()], page 252`[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 the 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);
```

This routine must add an object of class `NEW` to the current form. It receives 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 units (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,
```

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

```
                                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` and `my` contain the current mouse position and `key` the key that was pressed (if this information is related to the event). See Chapter 26 [Events], page 242, 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 is set for the object, whether the object is to be returned to the application program by `[fl_do_forms()], page 296` or `[fl_check_forms()], page 296`. 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()], page 124` and `[fl_get_button()], page 124`.

25.1 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 *obj;

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

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

    /* allocate instance-specific storage and fill it with defaults */
    obj->spec_size = sizeof SPEC;
```

```
        obj->spec = fl_calloc(1, obj->spec_size);
        return obj;
    }
```

The constant FL_NEW will indicate the object class. It should be an integer. The numbers 0 to FL_USER_CLASS_START - 1 (1000) and FL_BEGIN_GROUP (10000) and higher are reserved for the system and should not be used. Also it is preferable to use fl_malloc(), fl_calloc(), fl_realloc() and fl_free() to allocate/free the memory for the instance specific structures. These routines have the same prototypes and work the same way as those in the standard library and may offer additional debugging capabilities in future versions of the Forms Library. Also note that these functions are actually function pointers, and if desired, the application is free to assign these pointers to its own memory allocation routines.

There's also a version equivalent to the strdup() POSIX function which used [fl_malloc()], page 241:

```
    char * fl_strdup(const char *s);
```

The object pointer returned by [fl_make_object()], page 252 will have all of its fields set to some defaults (see Chapter 27 [The Type FL_OBJECT], page 247). 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()], page 252. Conversion of units, if different from the default pixel, is performed within [fl_make_object()], page 252 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 *obj;
        obj = fl_create_NEW(type, x, y, w, h, label);
        fl_add_object(fl_current_form, obj);
        return obj;
    }
```

26 Events

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

The following types of events can be sent to an object:

`FL_DRAW` The object has to be (re)drawn. To figure out the actual size of the object you can use the fields `obj->x`, `obj->y`, `obj->w` and `obj->h`. Many Xlib drawing routines require a window ID, which you can obtain from the object pointer using `FL_ObjWin(obj)`. 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 the following way: The field `obj->belowmouse` tells you 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 drawing of the object is the full responsibility of the object class, including the bounding box and the label, which can be found in the field `obj->label`. The Forms Library provides a large number of routines to help you draw object. See Chapter 28 [Drawing Objects], page 254, for more details on drawing objects and an overview of all available routines.

One important caution about your draw event handling code: none of the high level routines (`[fl_freeze_form()], page 289`, `[fl_deactivate_form()], page 296`) etc. can be used. The only routines allowed to be used are (direct) drawing functions and object internal book keeping routines. Attribute modifying routines, such as `[fl_set_object_color()], page 287` etc. are not allowed (using them can lead to infinite recursions). In addition, (re)drawing of other objects using `[fl_redraw_object()], page 297` while handling `[FL_DRAW], page 242` will also not work.

Due to the way double buffering is handled, at the time the `FL_DRAW` event is passed to the handling function (and only then) `FL_ObjWin(obj)` might return a pixmap used as the backbuffer (at least if the object is double buffered). What that means is that `FL_ObjWin(obj)` should not be used when a real window is needed. For a real window you can change the window's cursor or query the mouse position within it. You can't do either of these with the backbuffer pixmap. If there is a need to obtain the real window ID the following routine can be used:

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

To summarize: use `FL_ObjWin(obj)` when drawing and use `[fl_get_real_object_window()], page 242` for cursor or pointer routines. This distinction is important only while handling `FL_DRAW` events, `FL_ObjWin(obj)` should be used anywhere else.

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

FL_ENTER
: This event is sent when the mouse has entered the bounding box and might require some action. Note also that the field obj->belowmouse in the object is being set. If entering an objects area only changes its appearance, redrawing it normally suffices. Don't do this directly! Always redraw the object by calling [fl_redraw_object()], page 297. It will send an FL_DRAW event to the object but also does some other things (like setting window IDs and taking care of double buffering etc.).

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

FL_MOTION
: Motion events get sent between FL_ENTER and FL_LEAVE events when the mouse position changes on the object. The mouse position is given as an argument to the handle routine.

FL_PUSH
: The user has pushed a mouse button on the object. Normally this requires some actual action. The number of the mouse button pushed is given in the key parameter, having one of the following values:

  FL_LEFT_MOUSE, FL_MBUTTON1
  : Left mouse button was pressed.

  FL_MIDDLE_MOUSE, FL_MBUTTON2
  : Middle mouse button was pressed.

  FL_RIGHT_MOUSE, FL_MBUTTON3
  : Right mouse button was pressed.

  FL_SCROLLUP_MOUSE, FL_MBUTTON4
  : Mouse scroll wheel was rotated in up direction.

  FL_SCROLLDOWN_MOUSE, FL_MBUTTON5
  : Mouse scroll wheel was rotated in down direction.

FL_RELEASE
: The user has released the mouse button. This event is only sent if a [FL_PUSH], page 243 event was sent before. [FL_PUSH], page 243 event.

FL_DBLCLICK
: The user has pushed a mouse button twice within a certain time limit (FL_CLICK_TIMEOUT), which by default is 400 msec. 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 events from the scroll wheel.

FL_TRPLCLICK
: The user has pushed a mouse button three times within a certain time window. This event is sent after a [FL_DBLCLICK], page 243, [FL_PUSH], page 243,

[FL_RELEASE], page 243 sequence. Set click timeout to none-zero to activate FL_TRPLCLICK.

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

FL_UNFOCUS
: Input is no longer focussed on the object.

FL_KEYPRESS
: A key was pressed. The ASCII value (or KeySym if non-ASCII) is passed to the routine via the key argument, modifier keys can be retrieved from the state member of the XEvent also passed to the function via xev.

  This event only happens between [FL_FOCUS], page 244 and [FL_UNFOCUS], page 244 events. Not all objects are sent keyboard events, only those that have non-zero value in field obj->input or obj->wantkey.

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

FL_STEP
: A FL_STEP event is sent all the time (typically about 20 times a second but possibly less often because of system delays and other time-consuming tasks) to objects for which the field obj->automatic has been set to a non-zero value. The handling routine receives a synthetic MotionNotify event as the XEvent. This can be used to make an object change appearance without user action. Clock and timer objects use this type of event.

FL_UPDATE
: An FL_UPDATE event, like the [FL_STEP], page 244 event, also gets send about every 50 msec (but less often under high load) to objects while they are "pushed", i.e., between receiving a [FL_PUSH], page 243 and a [FL_RELEASE], page 243 event if their obj->want_update field is set. Like for the FL_STEP event the handling routine receives a synthetic MotionNotify event as the XEvent. This is typically used by objects that have to perform tasks at regular time intervals while they are "pushed" (e.g., counters that need to count up or down while the mouse is pushed on one of its buttons).

FL_ATTRIB
: An FL_ATTRIB event is sent to an object (via calling the handler function each object type must define for this purpose) whenever one of it's properties changes, be it its size, position, box type, border width, colors, label, label color, style or alignment etc. This can e.g., be used by the object to do preparations for later drawing of it or check that what got set is reasonable. It should not use this event to actually draw anything (this is to be done only when an [FL_DRAW], page 242 event is received). When the handler function for events is called all the arguments it gets passed are 0.

FL_FREEMEM
: This event is sent when the object is to be freed. All memory allocated for the object internally must 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 and 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()], page 297.

26.1 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 (i.e., not objects like boxes, texts etc.).

The mechanism works as follows. There is a routine

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

with which one 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 <Ctrl>h are associated with the object. The precise format is as follows: Any character in the string is considered as a shortcut, except '^' and '#', which stand for combinations with the <Ctrl> and <Alt> keys. (The case of the key following '#' or '^' is not important, i.e., no distiction is made between e.g., "^C" and "^c", both encode the key combination <Crl>C as well as <Crtl>C.) The key '^' itself can be set as a shortcut key by using "^^" in the string defining the shortcut. The key '#' can be obtained as a shortcut by using th string "^#". So, e.g., "#^#" encodes <ALT>#. The <Esc> key can be given as "^[".

Another special character not mentioned yet is '&', which indicates function and arrow keys. Use a sequence starting with '&' and directly followed by a number between 1 and 35 to represent one of the function keys. For example, "&2" stands for the <F2> function key. 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 as a shortcut by prefixing it with '^'.

The argument showit tells 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 the object label "foobar" the shortcut "oO" would result in a match at the first o in "foobar" while "Oo" would not. However, "^O" and "#O" would match since for keys used in combination with <Crtl> and <Alt> no distiction is made between upper and lower case.

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

```
void fl_set_object_shortcutkey(FL_OBJECT *obj, unsigned int key);
```

where key is an X KeySym, for example XK_Home, XK_F1 etc. Note that the function [fl_set_object_shortcutkey()], page 245 always appends the key specified to the current shortcuts while [fl_set_object_shortcut()], page 245 resets the shortcuts. Of course, special keys can't be underlined.

Now, whenever the user presses one of these keys, an [FL_SHORTCUT], page 244 event is sent to the object. The key pressed is passed to the handle routine (in the argument

key). Combinations with the <Alt> key are given by adding [FL_ALT_MASK], page 156 (currently the 25th bit, i.e., 0x1000000) to the ASCII value of the key. E.g., the key combinations <Alt>E and <Alt>e are passed as [FL_ALT_MASK], page 156, + 'E'. 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()], page 245 to register the shortcut in the event dispatching module. The idea is NOT that the user himself calls [fl_set_object_shortcut()], page 245 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 will suffice. For example [fl_set_button_shortcut()], page 124 is defined as [fl_set_object_shortcut()], page 245.

The order in which keys are handled is as follows: First for a key it is tested whether any object in the form has the key as a shortcut. If yes, 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], page 246 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 in the obj->wantkey field the [FL_KEY_TAB], page 246 bit is set. This is e.g., used in multi-line input fields. If the object wants all cursor keys (including <PgUp> etc.), the obj->wantkey field must have the [FL_KEY_SPECIAL], page 246 bit set.

To summarize, the obj->wantkey field can take on the following values (or the bit-wise or of them):

FL_KEY_NORMAL
: The default. The object receives left and right cursor, <Home> and <End> keys plus all normal keys (0-255) except <Tab> <Return>.

FL_KEY_TAB
: Object receives the <Tab>, <Return> as well as the <Up> and <Down> cursor keys.

FL_KEY_SPECIAL
: The object receives all keys with a KeySym above 255 which aren't already covered by FL_KEY_NORMAL and FL_KEY_TAB (e.g., function keys etc.)

FL_KEY_ALL
: Object receives all keys.

This way it is possible for a non-input object (i.e., if obj->input is zero) to obtain special keyboard event by setting obj->wantkey to [FL_KEY_SPECIAL], page 246.

27 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. Please always use accessor methods when available instead of using or changing the object's properties directly. Below we consider some of them that are likely to be used in new classes.

`int objclass`
: This indicates the class of the object (e.g., `FL_BUTTON`, `FL_SLIDER`, `FL_NEW` etc.) The user can query the class of an object using the function [`fl_get_object_objclass()`], page 286.

`int type`
: This indicates the type of the object within the class. Types are integer constants that should be defined in a header file named after the object class, e.g., `NEW.h`. Their use is completely free. For example, in the slider class 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. The user can query the type of an object using the function [`fl_get_object_type()`], page 286.

`int boxtype`
: This is the type of the bounding box for the object. The handling routine for the object, e.g., `handle_NEW()`, has to take care that this is actually drawn. Note that there is a routine for drawing boxes, see below. The user can change or query the boxtype of an object with the functions [`fl_set_object_boxtype()`], page 287 and [`fl_get_object_boxtype()`], page 287.

`FL_Coord x, y, w, h`
: These are the coordinates and sizes that indicate the bounding box of the object. They always have to be provided when adding an object. The system uses them e.g., to determine if the object is 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. The routines [`fl_get_object_geometry()`], page 287, [`fl_get_object_position()`], page 287 and [`fl_get_object_size()`], page 287 should be used to determine position and/or size. To change the position and/or size of an object never change the elements of the structures directly (except in a function like `fl_add_NEW()`) but always use [`fl_set_object_geometry()`], page 287, [`fl_set_object_position()`], page 287, [`fl_set_object_size()`], page 287 and [`fl_move_object()`], page 287!

  Also note that the `y`-member is always relative to the top of the form the object belongs to, even if the user had called [`fl_flip_yorigin()`], page 282 - this only results in `y`-values passed by and returned to the user when using functions like [`fl_set_object_position()`], page 287 or [`fl_get_object_position()`], page 287 getting "flipped", internally always the normal coordinate system is used.

`unsigned int resize`

Controls if the object should be resized if the form it is on is resized. The options are FL_RESIZE_NONE, FL_RESIZE_X, FL_RESIZE_Y and FL_RESIZE_ALL. The default is FL_RESIZE_ALL which is the bitwise OR of FL_RESIZE_X and FL_RESIZE_Y. Instead of accessing this element directly better use the functions [fl_get_object_resize()], page 289 and [fl_set_object_resize()], page 289.

`unsigned int nwgravity, segravity`

These two variables control how the object is placed relative to its position prior to resizing. Instead of accessing these elements directly use [fl_get_object_gravity()], page 289 and [fl_set_object_gravity()], page 289.

`FL_COLOR 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 change them using the routine [fl_set_object_color()], page 287 or inspect the colors with [fl_get_object_color()], page 287. The routine fl_add_NEW() should fill in defaults.

`char *label`

This is a pointer to an allocated text string. This can be used by class routines to provide a label for the object. The class routines may not forget to allocate storage for it when it sets the pointer itself, i.e., doesn't use [fl_set_object_label()], page 288 - an empty label should be the empty string and not just a NULL pointer. The user can change it using the routines [fl_set_object_label()], page 288 and [fl_set_object_label_f()], page 288 or ask for it using [fl_get_object_label()], page 288. The label must be drawn by the routine handling the object when it receives a FL_DRAWLABEL event (or it could be part of the code for FL_DRAW event). For non-offsetted labels, i.e., the alignment is relative to the entire bounding box, simply calling [fl_draw_object_label()], page 264 should be enough.

`FL_COLOR lcol`

The color of the label. The class routines can freely use this. The user can set it with [fl_set_object_lcolor()], page 288 and test it with [fl_get_object_lcolor()], page 288.

`int lsize` The size of the font used to draw the label. The class routines can freely use this. The user can set it with [fl_set_object_lsize()], page 288. and test it with [fl_get_object_lsize()], page 288.

`int lstyle`

The style of the font the label os drawn in, i.e., the number of the font in which it should be drawn. The class routines can freely use this. The user can set it with [fl_set_object_lstyle()], page 288 and test it with [fl_get_object_lstyle()], page 288.

`int 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], page 30, [FL_ALIGN_RIGHT], page 30, [FL_ALIGN_TOP], page 30, [FL_ALIGN_BOTTOM], page 30, [FL_ALIGN_CENTER], page 30,

[FL_ALIGN_LEFT_TOP], page 30, [FL_ALIGN_RIGHT_TOP], page 30, [FL_ALIGN_LEFT_BOTTOM], page 30 and [FL_ALIGN_RIGHT_BOTTOM], page 30. The value should be bitwise ORed with [FL_ALIGN_INSIDE], page 30 if the label will be within the bounding box of the object. The user can set this using the routine [fl_set_object_lalign()], page 288 and test it with [fl_set_object_lalign()], page 288.

int bw
: An integer indicating the border width of the object. Negative values indicate the up box should look "softer" (in which case no black line of 1 pixel width is drawn around the objects box). The user can set a different border width using [fl_set_object_bw()], page 287.

long *shortcut
: A pointer to long containing all shortcuts (as keysyms) defined for the object (also see the previous chapter). You should never need them because they are fully handled by the main routines.

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

int 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. The visisbility of an object can be tested using the [fl_object_is_visible()], page 290 function. Note that a this doesn't guarantee that the object is visible on the screen, for this also the form the object belongs to needs to be visible, in which case [fl_form_is_visible()], page 296 returns true.

int 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. This property should be set in the fl_add_NEW() routine if required. By default objects are active. This attribute can be changed by using the functions [fl_deactivate_object()], page 297 and [fl_activate_object()], page 297 and the current state can be determined by calling [fl_object_is_active()], page 297.

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

int wantkey
: An input object normally does not receive <Tab> or <Return> keystrokes or any other keys except those that have values between 0-255, the <Left> and <Right> arrow keys and <Home> and <End> (<Tab> and <Return> are normally

used to switch between input objects). By setting this field to FL_KEY_TAB enforces that the object receives also these two keys as well as the <Up> and <Down> arrow keys and <PgUp> and <PgDn> when it has the focus. To receive other special keys (e.g., function keys) FL_KEY_SPECIAL must be set in wantkey. By setting wantkey to FL_KEY_ALL all keys are sent to the object.

unsigned int click_timeout
: If non-zero this indicates the the maximum elapsed time (in msec) between two mouse clicks to be considered a double click. A zero value disables double/triple click detection. The user can set or query this value using the functions [fl_set_object_dblclick()], page 289 and [fl_get_object_dblclick()], page 289.

int automatic
: An object is automatic if it automatically (without user actions) has to change its contents. Automatic objects get a FL_STEP event about every 50 msec. For example the object class FL_CLOCK is automatic. automatic by default is false. To set this property use [fl_set_object_automatic()], page 289 (don't set the object member directly except from within a function like fl_add_NEW(), in other contexts some extra work is required) and to test the object for it use [fl_object_is_automatic()], page 289.

int 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.

int 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.

int 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.

FL_HANDLEPTR handle
: This is a pointer to the interaction handling routine for the object. fl_add_NEW() sets this by providing the correct handling routine. Normally it is never used (except by the main routine) 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.

FL_OBJECT *next, *prev
FL_FORM *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.

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

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

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

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

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

`long u_ldata`
: A long variable provided for the application program.

`FL_CALLBACKPTR object_callback`
: The callback routine that the application program assigned to the object and that the system invokes when the user does something with the object.

`long argument`
: The argument to be passed to the callback routine when invoked.

`int how_return`
: Determines under what circumstances the object is returned by e.g., [`fl_do_forms()`], page 296 or the callback function for the object is invoked. This can be either

  [`FL_RETURN_NONE`], page 45
  : Object gets never returned or its callback invoked

  [`FL_RETURN_CHANGED`], page 44
  : Return object or invoke callback when state of object changed.

  [`FL_RETURN_END`], page 44
  : Return object or invoke callback at end of interaction, normally when the mouse key is released or, in the case of input objects, the object has lost focus.

  [`FL_RETURN_END_CHANGED`], page 45
  : Return object or invoke callback only when interaction has ended and the state of the object has changed.

  [`FL_RETURN_SELECTION`], page 45
  : Return object or invoke callback if e.g., in a browser a line was selected.

  [`FL_RETURN_SELECTION`], page 45
  : Return object or invoke callback if e.g., in a browser a line was deselected.

[FL_RETURN_ALWAYS], page 45
: Return object or invoke callback whenever interaction has ended or the state of the object has changed.

Never change this element of the structure directly but use the function [fl_set_object_return()], page 44 instead! Especially in the case of objects having child objects also the corresponding settings for child objects may need changes and which automatically get adjusted when the above function is used.

int returned
: Set to what calling the object handling function did return (and pruned to what the object is supposed to return according to the how_return element). Can be either

[FL_RETURN_NONE], page 45
: Handling function did FL_RETURN_NONE (i.e., 0).

[FL_RETURN_CHANGED], page 44
: Handling function detected a change of the objects state.

[FL_RETURN_END], page 44
: Handling function detected end of interaction with object.

[FL_RETURN_CHANGED], page 44 and [FL_RETURN_END], page 44 are bits that can be bitwise ored. If both are set this indicates that the objects state was changed and the interaction ended.

The generic object construction routine

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

FL_OBJECT *fl_make_object(int objclass, int type,
                          FL_Coord x, FL_Coord y,
                          FL_Coord w, FL_Coord 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;
obj->lstyle     = FL_NORMAL_STYLE;
obj->col1       = FL_COL1;
obj->col2       = FL_MCOL;
obj->wantkey    = FL_KEY_NORMAL;
obj->active     = 1;
```

```
obj->visible    = 1;
obj->bw         = borderWidth_resource_set ? resource_val : FL_BOUND_WIDTH;
obj->u_ldata    = 0;
obj->u_vdata    = 0;
obj->spec       = NULL;
obj->how_return = FL_RETURN_CHANGED
```

In some situations it can be also useful to make an object a child of another object. An example is the scrollbar object. It has three child objects, a slider and two buttons, which all three are children of the scrollbar object. To make an object `child` a child object of an object named `parent` use the function

```
void fl_add_child(FL_OBJECT *parent, FL_OBJECT *child);
```

When creating a composite object you will typically add callbacks for the child object that handle what happens on events for these child objects (e.g., for the scrollbar the buttons have callbacks that update the internal state for the scrollbar object and result in the slider getting shifted). Within these callback functions the `returned` elements of the parent can be changed to influence if and what gets reported to the application via [fl_do_forms(), page 296.

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(), page 198 on the object to obtain the window resource ID of the window 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(), page 289.

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). Never change it directly, use [fl_show_form(), page 292 or [fl_hide_form(), page 295 instead.

`int deactivated`
: Indicates if the form is deactivated. Never change it directly, use [fl_activate_form(), page 296 or [fl_deactivate_form(), page 296 instead.

`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 forms window.

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

28 Drawing Objects

28.1 General Remarks

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 and size, i.e., the bounding box, of the object are indicated by the object tructure fields `obj->x`, `obj->y`, `obj->w` and `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. The 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 such structures, `fl_state[6]`, each for every visual class.

The structure contains among others the following members:

`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 does not 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]` A 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 `GC`s (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.

The currently active visual class (`TrueColor`, `PseudoColor` etc.) can be obtained by the following function/macro:

```
int fl_get_form_vclass(FL_FORM *form);
int fl_get_vclass(void);
```

The value returned can be used as an index into the array [fl_state], page 301 of [FL_State], page 254 structures. Note that [fl_get_vclass()], page 254 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 array. 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. The Forms Library comes with 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 [fl_display], page 301 can be used where required. However, it is recommended that you use fl_get_display() or FL_FormDisplay(Form *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 like FL_DRAW etc. belongs to. If the object's address is available, FL_ObjWin(obj) will suffice. Otherwise the function [fl_winget()], page 259 (see below) 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 or None on failure.

28.2 Color Handling

As mentioned earlier, Forms Library keeps an internal colormap, initialized to predefined colors. The predefined colors do not correspond to pixel values the server understands but are indexes into the colormap. Therefore, they can't be used in any of the GC altering or Xlib routines. To get the actual pixel value the X server understands, use the following routine

```
unsigned long fl_get_pixel(FL_COLOR col);
```

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

```
unsigned long red_pixel;
red_pixel = fl_get_pixel(FL_RED);
```

To change the foreground color in the Forms Library's default GC (gc[0]) use

```
void fl_color(FL_COLOR col);
```

To set the background color in the default GC use instead

```
void fl_bk_color(FL_COLOR col);
```

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 col);
```

```
void fl_set_background(GC gc, FL_COLOR col);
```

which is equivalent to the following Xlib calls

```
XSetForeground(fl_get_display(), gc, fl_get_pixel(color));
XSetBackground(fl_get_display(), gc, fl_get_pixel(color));
```

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 `n` colors stored in the array of colormap indices `cols`. You shouldn't do that for the reserved colors, i.e., colors with indices below `FL_FREE_COL1`.

In case the pixel values (instead of the index into the colormap) 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:

```
unsigned long fl_mapcolor(FL_COLOR col, int red, int green, int blue)
long fl_mapcolorname(FL_COLOR col, const char *name);
unsigned long fl_getmcolor(FL_COLOR col,
                           int *red, int *green, int *blue);
```

The first function, `[fl_mapcolor()], page 256` sets a the color indexed by `color` to the color given by the `red`, `green` and `blue`, returning the colors pixel value.

The second function, `[fl_mapcolorname()], page 256`, sets the color in the colormap indexed by `color` to the color named `name`, where `name` must be a valid name from the system's color database file `rgb.txt`. It also returns the colors pixel value or -1 on failure.

The last function, `[fl_getmcolor()], page 256`, returns the RGB values of the color indexed by `color` in the second to third argument pointers and the pixel value as the return value (or -1, cast to `unsigned long`, on failure).

28.3 Mouse Handling

The coordinate system used corresponds directly to that of the screen. But object coordinates are relative to the upper-left corner of the form the object belongs to.

To obtain the position of the mouse relative to a certain form or window, use the routines

```
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 ID of the window the mouse is in. Upon return `x` and `y` are set to the mouse position relative to the form or window and `keymask` contains information on modifier keys (same as the the corresponding `XQueryPointer()` argument).

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

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

The function returns the ID of the window the mouse is in.

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

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

Use this function sparingly, it can be extremely annoying for the user if the mouse position is changed by a program.

28.4 Clipping

To avoid drawing outside a 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` used for drawing (but not for output of text, see below). `x`, `y`, `w` and `h` define the area drawing is to restrict to and are relative to the window/form that will be drawn to. In this way you can prevent drawing over other objects.

Under some circumstances XForms also does it's own clipping, i.e., while drawing due to a exposure event. This is called "global clipping". Thus the clipping area you have set via a call of `[fl_set_clipping()], page 257` may get restricted even further due this global clipping.

You can check if there's clipping set for the default `GC` using the function

```
int fl_is_clipped(int include_global);
```

which returns `1` if clipping is switched on and `0` otherwise. The `include_global` argument tells the function if global clipping is to be included in the answer or not (i.e., if the argument is `0` only clipping set via `[fl_set_clipping()], page 257` is reported).

The area currently clipped to is returned by the function

```
int fl_get_clipping(int include_global, FL_Coord *x,FL_Coord *y,
                    FL_Coord *width, FL_Coord *height);
```

On return the four pointer arguments are set to the position and size of the clipping rectangle (at least if clipping is switched on) and the qreturn value of this function is the same as that of `[fl_is_clipped()], page 257`. The `include_global` argument has the same meaning as for `[fl_is_clipped()], page 257`, i.e., it controls if the effects of global clipping is included in the results.

When finished with drawing always use

```
void fl_unset_clipping(void);
```

to switch clipping of again.

You also can check and obtain the current settings for global clipping using the functions

```
int fl_is_global_clipped(void);
int fl_get_global_clipping(FL_Coord *x,FL_Coord *y,
                           FL_Coord *width, FL_Coord *height);
```

Clipping for text is controlled via a different `GC` and thus needs to be set, tested for and unset using a different set of functions:

```
void fl_set_text_clipping(FL_Coord x,FL_Coord y,FL_Coord w,FL_Coord h);
int fl_is_text_clipped(int include_global);
int fl_get_text_clipping(int include_global, FL_Coord *x,FL_Coord *y,
```

```
                                FL_Coord *width, FL_Coord *height);
void fl_unset_text_clipping(void);
```

Finally, there are functions to set and unset the clipping for a specific GC:

```
void fl_set_gc_clipping(GC gc, FL_Coord x, FL_Coord y,
                        FL_Coord width, FL_Coord height);
void fl_unset_gc_clipping(GC gc);
```

Please note that setting clipping for a GC will always further restrict the region to the region of global clipping (if it is on at the moment the function is called) and unsetting clipping will still retain global clipping if this is on at the moment the second function is invoked (if it is currently on can be checked using the [fl_is_global_clipped()], page 257).

28.5 Getting the Size

To obtain the bounding box of an object with the label taken into account (in contrast to the result of the [fl_get_object_geometry()], page 287, function which doesn't include a label that isn't inside the object the following routine exists:

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

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 *ascent, int *descent)
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. The first function, [fl_get_char_height()], page 258, also returns the height above and below the baseline of the font via the ascent and descent arguments (if they aren't NULL pointers). A list of valid styles can be found in Section 3.11.3.

To obtain the width and height information for 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 *ascent, int *descent);
```

where len is the length of the string str. The functions return the width and height of the string, respectively. The second function also returns the height above and below the fonts baseline if ascent and descent aren't NULL pointers. Note that the string may not contain newline characters '\n' and that the height calculated from the ascent and descent of those characters in the string that extend the most above and below the fonts baseline. It thus may not be suitable for calculating line spacings, for that use the [fl_get_char_height()], page 258 or [fl_get_string_dimension()], page 258 function.

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 characters '\n' and the routine will make proper adjustment so the values returned are large enough to contain the multiple lines of text. The height of each of the lines is the fonts height.

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

28.6 Font Handling

Sometimes it can be useful to get the X font structure for a particular size and style as used in the Forms Library. 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);
```

The caller is not allowed to free the structure returned by `[fl_get_fontstruct()], page 259`, it's just a pointer to an internal structure!

28.7 Drawing Functions

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. If that's not what you need, the following routines can be used to set or query the current window:

```
void fl_winset(Window win);
Window fl_winget(void);
```

One caveat about `[fl_winget()], page 259` is that it can return `None` if called outside of an object's event handler, depending on where the mouse is. Thus, the return value of this function should be checked when called outside of an object's event handler.

It is important to remember that unless the following drawing commands are issued while handling the `FL_DRAW` or `FL_DRAWLABEL` event (which is not generally recommended), it is the application's responsibility to set the proper drawable using `[fl_winset()], page 259`.

The most basic drawing routines are for drawing rectangles:

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

Both functions draw a rectangle on the screen in color `col`. While `[fl_rectf()], page 259` draws a filled rectangle, `[fl_rect()], page 259` just draws the outline in the given color.

To draw a filled (with color `col`) rectangle with a black border use

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

To draw a rectangle with rounded corners (filled or just the outlined) employ

```
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()], page 259 draws a filled polygon defined by `n` points, [fl_polyl()], page 259 the ouline of a polygon and [fl_polybound()], page 259 a filled polygon with a black outline.

Note: **all** polygon routines require that the array `xpoint` has spaces for `n+1` points, i.e., one more than then number of points you intend to draw!

To draw an ellipse. either filled, open (with the outline drawn in the given color), or filled with a black border the following routines can be used (use `w` equal to `h` to get a circle):

```
void fl_ovalf(FL_Coord x, FL_Coord y, FL_Coord w, FL_Coord h,
              FL_COLOR col);
void fl_ovall(FL_Coord x, FL_Coord y, FL_Coord w, FL_Coord h,
              FL_COLOR col);
void fl_ovalbound(FL_Coord x, FL_Coord y, FL_Coord w, FL_Coord h,
                  FL_COLOR col);
```

The `x` and `y` arguments are the upper left hand corner of the ellipse, while `w` and `h` are its width and height.

Note: [fl_ovall()], page 260 (with two 'l') isn't a typo, the trailing 'l' it's meant indicate that only a line will be drawn. And there's also the function

```
void fl_ovalf(int fill, FL_Coord x, FL_Coord y, FL_Coord w,
              FL_Coord h, FL_COLOR col);
```

which is invoked by both (the macros) [fl_ovalf()], page 260 and [fl_ovall()], page 260 with the first argument `fill` set to either `1` or `0`.

To simplify drawing circles there are three additional functions. The first one draws an (open) circle (with the circumfence in the given color), the second one a filled circle, and the last one a filled circle with a black circumfence:

```
void fl_circ(FL_Coord x, FL_Coord y, FL_Coord r, FL_COLOR col);
void fl_circf(FL_Coord x, FL_Coord y, FL_Coord r, FL_COLOR col);
void fl_circbound(FL_Coord x, FL_Coord y, FL_Coord r, FL_COLOR col);
```

Here `x` and `y` are the coordinates of the center of the circle, `r` is its radius and `col` the color to be used.

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 start_theta, int end_theta, FL_COLOR col);
```

`x` and `y` are the coordinates of the center and `r` is the radius. `start_theta` and `end_theta` are the starting and ending angles of the arc in units of tenths of a degree (where 0 stands for a direction of 3 o'clock, i.e., the right-most point of a circle), and `x` and `y` are the center of the arc. If the difference between `theta_end` and `theta_start` is larger than 3600 (360 degrees), drawing is truncated to 360 degrees.

To draw elliptical arcs the following routine can be used:

```
void fl_pieslice(int fill, FL_Coord x, FL_Coord y, FL_Coord w,
                 FL_Coord h, int start_theta, int end_theta,
                 FL_COLOR col);
```

`x` and `y` are the upper left hand corner of the box enclosing the ellipse that the pieslice is part of and `w` and `h` the width and height of that box. `start_theta` and `end_theta`, to be given in tenth of a degree, specify the starting and ending angles measured from zero degrees (3 o'clock).

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

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

Here `theta` specifies the starting angle (again measured in tenth of a degree and with 0 at the 3 o'clock position), and `dtheta` specifies both the direction and extent of the arc. If `dtheta` is positive the arc is drawn in counter-clockwise direction from the starting point defined by `theta`, otherwise in clockwise direction. If `dtheta` is larger than 3600 it is truncated to 3600.

To connect two points with a straight line, use

```
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 set `h` to 1, not to 0):

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

To draw connected line segments between `n` points use

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

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

There are also routines to draw one or more pixels

```
void fl_point(FL_Coord x, FL_Coord y, FL_COLOR col);
void fl_points(FL_POINT *p, int np, FL_COLOR col);
```

As usual, all coordinates are relative to the origin of the drawable. Note that these routines are meant for you to draw a few pixels, not images consisting of tens of thousands of pixels of varying colors. For that kind of drawing `XPutImage(3)` should be used. Or better yet, use the image support in the Forms Library (see Chapter 37 [Images], page 319). Also it's usually better when drawing multiple points to use fl_points(), even if that means that the application program has to pre-sort and group the pixels of the same color.

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

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

Set `lw` to 0 to reset the line width to the servers default. Line styles can take on the following values (also see `XChangeGC(3)`)

`FL_SOLID` Solid line. Default and most efficient.

`FL DOT` Dotted line.

`FL DASH` Dashed line.

`FL DOTDASH`
Dash-dot-dash line.

`FL LONGDASH`
Long dashed line.

`FL USERDASH`
Dashed line, but the dash pattern is user definable via [fl_dashedlinestyle()], page 262. Only the odd numbered segments are drawn with the foreground color.

`FL USERDOUBLEDASH`
Similar to `FL_LINE_USERDASH` but both even and odd numbered segments are drawn, with the even numbered segments drawn in the background color (as set by [fl_bk_color()], page 255).

The following routine can be used to change the dash pattern for `FL_USERDASH` and `FL USERDOUBLEDASH`:

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

Each element of the array `dash` is the length of a segment of the pattern in pixels (0 is not allowed). Dashed lines are drawn as alternating segments, each with the length of an element in `dash`. Thus the overall length of the dash pattern, in pixels, is the sum of all elements of `dash`. When the pattern is used up but the line to draw is longer it used from the start again. The following example code specifies a long dash (9 pixels) to come first, then a skip (3 pixels), a short dash (2 pixels) and then again a skip (3 pixels). After this sequence, the pattern repeats.

```
char ldash_sdash[] = {9, 3, 2, 3};
fl_dashedlinestyle(ldash_sdash, 4);
```

If `dash` is `NULL` or `ndashes` is `0` (or the `dash` array contains an element set to `0`) a default pattern of 4 pixels on and 4 fixels off is set.

It is important to remember to call [fl_dashedlinestyle()], page 262 whenever `FL_USERDASH` is used to set the dash pattern, otherwise whatever the last pattern was will be used. To use the default dash pattern you can pass `NULL` as the dash parameter to [fl_dashedlinestyle()], page 262.

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

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

There are 16 different possible settings for `mode` (see a Xlib programming manual for all the gory details). A of the more useful ones are

`GXcopy` Default overwrite mode. Final pixel value = Src

`GXxor` Bitwise XOR (exclusive-or) of the pixel value to be drawn with the pixel value already on the screen. Useful for rubber-banding.

`GXand` Bitwise AND of the pixel value to be drawn with the pixel value already on the screen.

`GXor` Bitwise OR of the pixel value to be drawn with the pixel value already on the screen.

`GXinvert` Just invert the pixel values already on the screen.

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

```
int fl_get_linewidth(void);
int fl_get_linestyle(void);
int fl_get_drawmode(void);
```

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

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

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

There is also a routine for drawing 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 col, int style, int size,
                 const char *str);
void fl_drw_text_beside(int align, FL_Coord x, FL_Coord y,
                        FL_Coord w, FL_Coord h, FL_COLOR col,
                        int style, int size, const char *str);
```

where `align` is the alignment, namely, `FL ALIGN LEFT`, `FL ALIGN CENTER` etc. `x`, `y`, `w` and `h` indicate the bounding box, `col` is the color of the text, `size` is the size of the font to use (in points) and `style` is the font style to be used (see Section 3.11.3 [Label Attributes and Fonts], page 27, for valid styles). Finally, `str` is the string itself, possibly containing embedded newline characters.

`[fl_drw_text()], page 263` draws the text inside the bounding box according to the alignment requested while `[fl_drw_text_beside()], page 263` draws the text aligned outside of the box. These two routines interpret a text string starting with the character `@` differently in drawing some symbols instead. Note that `[fl_drw_text()], page 263` puts a padding of 5 pixels in vertical direction and 4 in horizontal around the text. Thus the

bounding box should be 10 pixels wider and 8 pixels higher than required for the text to be drawn.

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 col,
                        int style, int size, char *str,
                        int FL_COLOR ccol, int pos);
```

where `ccol` is the color of the cursor and `pos` is its position which indicates the index of the character in `str` before which to draw the cursor (-1 means show no cursor). This routine does no interpretion of the special character `@` nor does it add padding around the text.

Given a bounding box and the size of an object (e.g., a label) to draw, the following routine can be used to obtain the position of where to draw it with a certain alignment and including padding:

```
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`. `obj_xsize` and `obj->ysize` are the width and height of the object to be drawn and `xmargin` and `ymargin` is the additional padding to use. `xpos` and `ypos` return the position to be used for drawing the object.

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

```
void fl_draw_object_label(FL_OBJECT *obj)
void fl_draw_object_label_outside(FL_OBJECT *obj);
```

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

An important aspect of (re)drawing an object is efficiency which can result in flicker and non-responsiveness if not handled with care. For simple objects 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 drawing is not to draw if you don't 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/update only those parts that need to.

29 An Example

Let us work through an example of how to create a simple object class named `colorbox`. Assume that 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 got pushed.

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

```
/* Class number must be between FL_USER_CLASS_START
    and FL_USER_CLASS_END */

#define FL_COLBOX        (FL_USER_CLASS_START + 1)

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

Note that the type must start from zero onward. Normally it should also contain some defaults for the boxtype and label alignment etc. The include file also has to declare all the functions available for this object class. I.e., it should contain:

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

Now 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;         /* no. of times pushed */
} COLBOX_SPEC;

FL_OBJECT *fl_create_colbox(int type, FL_Coord x, FL_Coord y,
                            FL_Coord w, FL_Coord h,
                            const char *label) {
    FL_OBJECT *obj;

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

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

    /* create class specific structures and initialize */
```

```
    obj->spec = fl_malloc(sizeof *obj->spec);
    obj->spec->counter = 0;
    return obj;
}

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

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()`], page 287. 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 `obj->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 *obj, int event,
                         FL_Coord mx, FL_Coord my,
                         int key, void *xev) {
    switch (event) {
        case FL_DRAW:              /* Draw box */
            fl_drw_box(obj->boxtype, obj->x,obj->y, obj->w, obj->h,
                       obj->pushed ? obj->col2 : obj->col1, obj->bw);
            /* fall through */

        case FL_DRAWLABEL:         /* Draw label */
             fl_draw_object_label(obj);
             break;

        case FL_PUSH:
            ((COLBOX_SPEC *) obj->spec)->counter++;
            fl_redraw_object(obj);
            break;

        case FL_RELEASE:
            fl_redraw_object(obj);
            return 1;             /* report back to application! */

         case FL_FREEMEM:
             fl_free(obj->spec);
             break;
    }
```

```
    return 0;
}
```

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

```
int fl_get_colbox(FL_OBJECT *obj) {
    if (!obj || obj->objclass != FL_COLBOX) {
        fprintf(stderr, "fl_get_colbox: Bad argument or wrong type);
        return -1;
    }

    return ((COLBOX_SPEC *) obj->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 with `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.

30 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()], page 124 and [fl_set_button()], page 124 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 is

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

which 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()], page 269, discussed below, can be used to register a drawing routine that completes the creation of a new button.

All button or button-like objects have 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 int   bits_w,   /* for bitmap/pixmap button only */
                   bits_h;
    int            val;      /* whether it's pushed */
    int            mousebut; /* mouse button that caused the push */
    int            timdel;   /* time since last touch (TOUCH buttons)*/
    int            event;    /* what event triggered the redraw */
    long           cspecl;   /* for non-generic class specific data */
    void         * cspec;    /* for non-generic class specific data */
    char         * file;     /* filename for the pixmap/bitmap file */
} FL_BUTTON_STRUCT;
```

Of all its 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 outweights the compromise on data hiding.

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

After the header file is created, you will have to write C functions that create and draw the button. You also will 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 *), void
(*cleanup)(FL_BUTTON_SPEC *));
```

where objclass is the class ID, and draw is a function that will be called to draw the button. cleanup is a function that will be called prior to destroying the button. You need a cleanup function only if the drawing routine uses the cspecv field of FL_BUTTON_STRUCT to hold memory allocated dynamically 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, i.e., its real working source code that implements the button class. To illustrate the entire process of creating this class, let us call this button class FL_NBUTTON.

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

```
#ifndef NBUTTON_H_
#define NBUTTON_H_

#define FL_NBUTTON  FL_USER_CLASS_START

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

#endif
```

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

```
static void draw_nbutton(FL_OBJECT *obj) {
    FL_COLOR col;

    /* box color. If pushed we use obj->col2, otherwise use obj->col1 */
    col = ((FL_BUTTON_STRUCT *) obj->spec)->val ?
          obj->col2 : obj->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 (obj->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 (   obj->boxtype == FL_UP_BOX
             && ((FL_BUTTON_STRUCT *) obj->spec)->val)
             fl_drw_box(FL_DOWN_BOX, obj->x, obj->y, obj->w, obj->h,
                        col, obj->bw);
         else
             fl_drw_box(obj->boxtype, obj->x, obj->y, obj->w, obj->h,
                        col, obj->bw);

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

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

Note that when drawing symbols, the style and size are irrelevent and set to zero in [`fl_drw_text()`], page 263 above.

Since we don't use the `cspecv` field to point to dynamically allocated memory 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 *obj;

        obj = fl_create_generic_button(FL_NBUTTON, type, x, y, w, h, label);
        fl_add_button_class(FL_NBUTTON, draw_nbutton, NULL);

        obj->col1  = FL_COL1;          /* normal color */
```

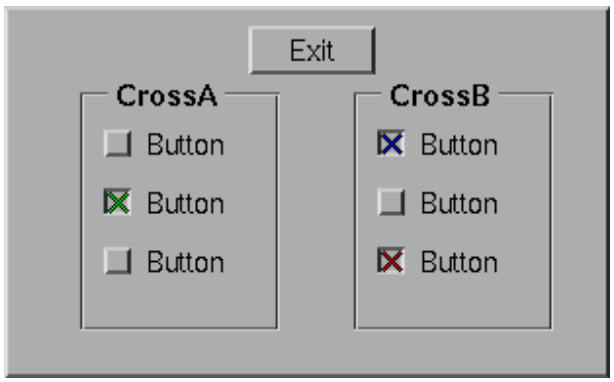
[1] A separate creation routine is useful for integration into the Form Designer.

```
    obj->col2  = FL_MCOL;          /* pushed color */
    obj->align = FL_ALIGN_CENTER;  /* button label placement */

    return obj;
}
```

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 *obj = fl_create_nbutton(type, x, y, w, h, label);

    fl_add_object(fl_current_form, obj);
    return obj;
}
```

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 a 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 the object class developers.



We choose to use `obj->col1` as the color of the box and `obj->col2` as the color of the cross (remember these two colors are changeable by the application program via `[fl_set_object_color()], page 287`). Note that this decision on color use is somewhat arbitrary, we could have easily made `obj->col2` the color of the button when pushed and use `obj->spec->cspecl` for the cross color (another routine named e.g., `fl_set_crossbutton_crosscol()` should be provided to change the cross color in this case).

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

```
#ifndef CROSSBUTTON_H_
#define CROSSBUTTON_H_

#define FL_CROSSBUTTON (FL_USER_CLASS_START + 2)

extern FL_OBJECT *fl_add_crossbutton(int, FL_Coord, FL_Coord,
```

```
                                        FL_Coord, FL_Coord, const char *);

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

Next we write the actual code that implements crossbutton class and put it into `crossbut.c`:

```
/* routines implementing the "crossbutton" class */

#include <forms.h>
#include "crossbut.h"

/** How to draw it */

static void draw_crossbutton(FL_OBJECT *obj) {
    FL_Coord xx, yy, ww, hh;
    FL_BUTTON_STRUCT *sp = obj->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(obj->boxtype, obj->x, obj->y, obj->w, obj->h,
               obj->col1, obj->bw);

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

    /* If pushed, draw a down box with the cross */
    if (sp->val) {
        fl_drw_box(FL_DOWN_BOX, xx, yy, ww, hh, obj->col1, obj->bw);
        fl_drw_text(FL_ALIGN_CENTER, xx - 2, yy - 2, ww + 4, hh + 4,
                    obj->col2, 0, 0, "@9plus");
    } else
        fl_drw_box(FL_UP_BOX, xx, yy, ww, hh, obj->col1, obj->bw);

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

```
    if (obj->type == FL_RETURN_BUTTON)
        fl_drw_text(FL_ALIGN_CENTER, obj->x + obj->w - 0.8 * obj->h,
                    obj->y + 0.2 * obj->h, 0.6 * obj->h, 0.6 * obj->h,
                    obj->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.

The 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 *obj;

    fl_add_button_class(FL_CROSSBUTTON, draw_crossbutton, NULL);

    /* 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;
     */

     obj = fl_create_generic_button(FL_CROSSBUTTON, type, x, y, w, h,
                                    label);
     obj->boxtype = FL_NO_BOX;
     obj->col2 = FL_BLACK; /* cross color */

     return obj;
}

/* 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 *obj = fl_create_crossbutton(type, x, y, w, h, label);

   fl_add_object(fl_current_form, obj);
   return obj;
}
```

The actual code is in the demo directory, see the files crossbut.c and 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 by including the actual header). You can also place the object file (`crossbut.o`) in `libforms.a` and `libforms.so` if you wish. Note however that this will make your application programs dependent on your personal version of the library.

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 the 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 `newbutton.c`.

31 Using a Pre-emptive Handler

Pre-emptive handlers came 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.

As mentioned earlier, an object module communicates with the main module via events. Central part of the module is the event handler, which determines how an object responds to various events such as mouse clicks or a key presses. Now a pre-emptive handler is a function 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 (and a possibly also installed post handler), thus altering the behavior of the object. A 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 *obj, int event,
                            FL_Coord mx, FL_Coord my,
                            int key, void *raw_event);

void fl_set_object_prehandler(FL_OBJECT *obj,
                              FL_HANDLEPTR pre_handler);
void fl_set_object_posthandler(FL_OBJECT *obj,
                               FL_HANDLEPTR post_handler);
```

`event` is a generic event of the Forms Library, that is, `[FL_DRAW], page 242`, `[FL_ENTER], page 243` etc. Parameters `mx` and `my` are the mouse position and `key` is the key pressed. The last parameter `raw_event` is a pointer to the XEvent (cast to a void pointer due to the different types of Xevents) that caused the invocation of the pre- or post-handler. But note: not all events of the Form Library have a corresponding Xevent and thus dereferencing of `xev` should only be done after making sure it is not `NULL`.

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 pre-handler should return `FL_PREEMPT` to prevent the dispatcher from calling the normal object handler for events and `!FL_PREEMPT` if the objects handler for is to be invoked next. The post-handler may return whatever it wants since 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], page 242`) receives all events.

Note that when an object has been de-activated using `[fl_deactivate_object()], page 297` (or the whole form the object belongs to is de-activated via calls of `[fl_deactivate_form()], page 296` or `[fl_deactivate_all_forms()], page 296`) also pre-emptive and post-handlers won't get invoked for the object.

See the demo programs `preemptive.c` and `xyplotall.c` for examples. Bear in mind that modifying the built-in behavior is in general not a good idea. Using a pre-emptive handler for the purpose of "peeking", however, is quite legitimate and can be useful in some situations.

Part V - General Informations

32 Overview of Main Functions

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

32.1 Version Information

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

`FL_VERSION`
: The major version number.

`FL_REVISION`
: Revision number.

`FL_INCLUDE_VERSION`
: `1000 * FL_VERSION + 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 `1000 * version + revision`. For example, for library version 1 revision 21 (1.21), the function returns a value of 1021 with `version` and `revision` (if not `NULL`) set to 1 and 21, respectively.

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

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

32.2 Initialization

The routine

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

initializes the Forms Library and returns a pointer to the `Display` structure if a connection could be made, otherwise `NULL`. This function must be called before any other calls to the Forms Library (except `[fl_set_defaults()], page 279` and a few other functions that alter some of the defaults of the library).

The meaning of the arguments is as follows

`argc, argv`
: Number and array of the command line arguments the application was started with. The application name is derived from `argv[0]` by stripping leading path names and trailing period and extension, if any. Due to the way the X resources (and command line argument parsing) work, the executable name should not contain a dot `.` or a star `*`.

`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()], page 277` function builds the resource database, calls the Xlib `XrmParseCommand()` function to parse the command line arguments and performs other per display initialization. After the creation of the database, it is associated with the display via `XrmSetDatabase()`, so the application can get at it if necessary.

All recognized options are removed from the argument list and their corresponding values set. The XForms library provides appropriate defaults for all options. The following are recognized by the library:

Option	**Type**	**Meaning**	**Default**
`-fldebug` *level*	int	Print debug information	0 (off)
`-name` *appname*	string	Change application name	none
`-flversion`		Print version of the library	
`-sync`		Synchronous X11 mode (debug)	false
`-display` *host:dpy*	string	Set (remote) host	`$DISPLAY`
`-visual` *class*	string	TrueColor, PseudoColor...	best
`-depth` *depth*	int	Set prefered visual depth	best
`-vid` *id*	long	Set prefered visual ID	0
`-private`		Force use of private colormap	false
`-shared`		Force use of shared colormap	false
`-stdcmap`		Force use of standard colormap	false
`-double`		Enable double buffering for forms	false
`-bw` *width*	int	Set object border width	1
`-rgamma` *gamma*	float	Set red gamma	1.0
`-ggamma` *gamma*	float	Set green gamma	1.0

`-bgamma` *gamma*	float	Set blue gamma	1.0

In the above table "best" means the visual that has the most colors, which may or may not be the server's default. There is a special command option `-visual Default` that sets both the visual and depth to the X servers default. If a visual ID is requested, it overrides depth or visual if specified. The visual ID can also be requested programmatically (before `[fl_initialize()], page 277` is called) via the function

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

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

If the border width is set to a negative number, all objects appear to have a softer appearance. Older version of the library used a larger default for the border width of 3.

As mentioned the `[fl_initialize()], page 277` function removes all the above listed values from the command line arguments, leaving you with a cleaned-up list. To get again at the complete list you can use the function

```
char **fl_get_cmdline_args( int *arg_cnt );
```

returning a copy to the values from the original list and their number via the `arg_cnt` argument.

Depending on your application XForms defaults may or may not be appropriate. E.g., on machines capable of 24 bits visuals, Forms Library always selects the deeper 24 bits visual. If your application only uses a limited number of colors, it might be faster if a visual other than 24 bits 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 defaults programmatically without affecting the users' ability to override them with command line options. For this, you can use the following routine before calling `[fl_initialize()], page 277`:

```
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.

The following table lists the fields, masks and their meanings of `[FL_IOPT], page 279`:

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`	Prefered visual, `TrueColor` etc.
`int doubleBuffer;`	`FL_PDDouble`	Simulate double buffering

`int buttonFontSize;`	`FL_PDButtonFontSize`	Default button label font size
`int menuFontSize;`	`FL_PDMenuFontSize`	Menu label font size
`int choiceFontSize;`	`FL_PDChoiceFontSize`	Choice label and choice text font size
`int browserFontSize;`	`FL_PDBrowserFontSize`	Browser label and text font size
`int inputFontSize;`	`FL_PDInputFontSize`	Input label and text font size
`int labelFontSize;`	`FL_PDLabelFontSize`	Label font size for all other objects (box, pixmap etc.)
`int pupFontSize;`	`FL_PDPupFontSize`	Font size for pop-ups
`int privateColormap;`	`FL_PDPrivateMap`	Select private colormap if appropriate
`int sharedColormap;`	`FL_PDSharedMap`	Force use of shared colormap
`int standardColormap;`	`FL_PDStandardMap`	Force use of standard colormap
`int scrollbarType;`	`FL_PDScrollbarType`	Scrollbar type to use for browser and input
`int ulThickness;`	`FL_PDULThickness`	Underline thickness
`int ulPropWidth;`	`FL_PDULPropWidth`	Underline width, 0 for const. width fonts
`int backingStore;`	`FL_PDBS`	Turn BackingStore on or off
`int coordUnit;`	`FL_PDCoordUnit`	Unit of measure: pixel, mm, point
`int borderWidth;`	`FL_PDBorderWidth`	Default border width

`} FL IOPT;`

A special visual designation, `FL_DefaultVisual` and a 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 an arbitrary name that is specified with the `-name` option.

Coordinate units can be in pixels, points (1/72 inch), mm (millimeters), cp (centi-point, i.e., 1/100 of a point) or cmm (centi-millimeter). The the type of unit in use can be queried or set via the functions

```
int fl_get_coordunit(void);
void fl_set_coordunit(int coordUnit);
```

`coordUnit` can have the following values: `FL_COORD_PIXEL`, `FL_COORD_POINT`, `FL_COORD_MM`, `FL_COORD_centiPOINT` and `FL_COORD_centiMM`.

The unit in use 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);
```

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 fonts and 2 for bold fonts.

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

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

which is equivalent to

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

Typically this function, if used, should appear before `[fl_initialize()], page 277` is called so the user has the option to override the default via resource or command line options.

The cirrent setting of the borderwidth can also tested via

```
int fl_get_border_width(void);
```

To change the default scrollbar type (which is `THIN_SCROLLBAR`) used in browser and input object, the following convenience function can be used:

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

where `type` can be one of the following

`FL_NORMAL_SCROLLBAR`
: Basic scrollbar

`FL_THIN_SCROLLBAR`
: Thin scrollbar

`FL_NICE_SCROLLBAR`
: Nice scrollbar

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

Setting the scrollbar type before calling `[fl_initialize()], page 277` is equivalent to

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

It is recommended that this function be used before `[fl_initialize()], page 277` so the user has the option to override the default through application resources.

Prior to version 0.80 the origin of XForms' coordinate system was at the lower left-hand corner of the form. The new Form Designer will convert the form definition file to the new coordinate system, i.e., with the origin at the upper left-hand corner, so no manual intervention is required. To help those who lost the `.fd` files or otherwise can't use a newer version of `fdesign`, a compatibility function is provided

```
void fl_flip_yorigin(void);
```

Note however that this function must be called prior to `[fl_initialize()], page 277` and is a no-op after that.

If this function has been called functions like `[fl_get_object_position()], page 287` or `[fl_get_object_bbox()], page 287`, reporting an objects positions and bounding box, will return `y`-coordinates in the old-fashioned coordinate system with the origin at the left bottom corner of the form. Similarly, the functions for setting or changing an objects position (`[fl_set_object_position()], page 287` and `[fl_move_object()], page 287`) then expect to receive arguments for the `y`-coordinates in this system. The `y`-coordinate stored in the object itself (i.e., `obj->y`) is always for the normal coordinate system with the origin at the top left corner.

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 `"aaaaaaaa"`, i.e., eight `'a'`s.

Before we proceed further, some comments about double buffering are in order. Since Xlib does not support double buffering, Forms Library simulates this functionality with pixmap bit-bliting. 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 a 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_no);
void fl_set_object_dblbuffer(FL_OBJECT *obj, int yes_no);
```

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

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

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

```
                                                const char *fmt,...));
```

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

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

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

For example

```
fl_set_error_logfp(fopen("/dev/null","w"));
```

redirects all error messages to `/dev/null`, effectively turning off the default error reporting to `stderr`.

In XForms versions older than 1.0.01 for some error messages, in addition to being printed to stderr, a dialog box were shown that requires actions from the user. This could be turned off and on with the function

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

where `show` indicates whether to show (1) or not show (0) the errors. With newer versions of the Forms Library this function has no effect.

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

```
int fl_set_font_name(int n, const char *name);
int fl_set_font_name_f(int n, const char *fmt, ,,,);
```

The first function just accepts a simple string while the second constructs the font name from a format string just as it's used for `printf()` etc. and the following arguments. The first argument, `n`, must be a number between 0 and `FL_MAXFONTS-1`. The function returns `0` on success, `1` if called before proper initialization of the library and `-1` for either invalid arguments (`name` or the result of the expansion of the format string doesn't name an available font, `n` negative or not less than `FL_MAXFONTS`). See Section 3.11.3 [Label Attributes and Fonts], page 27, for details. A redraw of all forms is required to actually see the change for visible forms.

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

Using a resolution independent unit for the object size should solve the font problems, theoretically. In practice, this approach may still prove to be vulnerable. The reason is the discreteness of both the font resolution and the monitor/server resolutions. The standard X fonts only come in two discrete resolutions, 75 DPI and 100 DPI. Due to the variations in monitor resolutions, the theoretically identical sized font, say a 10 pt font, can vary in sizes (pixels) by up to 30%, depending on the server (rendering a font on a 80 DPI monitor will cause errors in sizes regardless if a 75 DPI or 100 DPI font is used.) This has not even

taken into account the fact that a surprising number of systems have wrong font paths (e.g., a 90 DPI monitor using 75 DPI fonts etc.).

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

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

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

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

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

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

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

All colors with indices smaller than `FL_FREE_COL1` are used (or can potentially be used) by the Forms Library. If you wish they can be changed using the following function prior to `[fl_initialize()], page 277`:

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

Using this function you can actually change all entries in the internal colormap (with `index` going up to `FL_MAX_COLORS-1`). You may also inspect the internal colormap using

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

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 the application exits, all server defaults are restored. The only exception is when exiting from

a callback that is activated by shortcuts. Thus it is recommended that the cleanup routine [fl_finish()], page 285 is called prior to exiting an application or register it via atexit().

```
void fl_finish(void);
```

In addition to restoring all server defaults, [fl_finish()], page 285 also shuts down the connection and frees dynamically allocated memory.

32.3 Creating Forms

To start the definition of a form call

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

When the form is created it automatically acquires one object, a box object covering the full area of the form, which is used as the background of the form. The type argument is the type of this box object, so you can "style" the look of your forms (but don't use any non-rectangular box types). w and h are the width and height of the new form. The function returns a pointer to the new form.

Note: if you look at the code generated by fdesign for the creation of a form you may notice that the type of this automatically assigned box is [FL_NO_BOX], page 109 (which is invisible) and that for the background another box of the same size but a different (visible) type is added. This is because in fdesign the very first object can't be accessed and thus its properties can not be adjusted (like the box type or its color that then becomes the background color of the form). By using an extra box, which can be accessed from within fdesign, that problem is circumvented.

There also exist functions for setting and requesting the background color of a form

```
void fl_set_form_background_color(FL_FORM *form, FL_COLOR col);
FL_COLOR fl_get_form_background_color(FL_FORM *form);
```

These functions use the color of the very first object of the form, or, if this is a box of type [FL_NO_BOX], page 109 as it is the case with forms created via code generated by fdesign, the color of the second object. If these object(s) don't exist the function can't work properly.

Once all objects required have been added to a form call

```
void fl_end_form(void);
```

Between these two calls objects and groups of objects are added to the form with functions like [fl_add_button()], page 120.

To start a new group of objects use

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

The function returns a pointer to the group (actually to an invisible pseudo-object of class FL_BEGIN_GROUP). Groups can't be nested.

When all objects that are supposed to belong to the group are added call

```
void fl_end_group(void);
```

Also this function creates an (invisible) pseudo-object, belonging to class FL_END_GROUP, but since it can't be used its address isn ot returned.

Groups are useful for two reasons. First of all, it is possible to hide or deactivate groups of objects with a single function call. 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 will suffice.

The second reason for use of groups is 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.

Both forms and groups that have been ended by [fl_end_form()], page 285 or [fl_end_group()], page 285 can be "reopened" by using

```
FL_FORM *fl_addto_form(FL_FORM *form)
FL_OBJECT *fl_addto_group(FL_OBJECT *group);
```

Both functions return their argument on success and NULL on failure (e.g., because a different group or form is still open). On success further objects can be appended to the form or group.

To remove an object from a form use

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

This does not yet destroy the object, it just breaks its connection to the form it did belong to, so it can still be referenced and added to the same form again or some other form using

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

even without "reopening" the form using [fl_addto_form()], page 286.

To finally destroy an object use

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

If [fl_delete_object()], page 286 hadn't been called for the object this will happen now. The object receives a final event of type [FL_FREEMEM], page 244 to allow it to free memory it did allocate and do whatever other clean-up required. Finally all memory allocated for the object is freed. After being freed an object can not be referenced anymore.

A form as a whole, together with all the objects it contains can be deleted by calling

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

This will first hide the form (emitting warning if this is necessary), then free all of its objects and finally release memory allocated for the form.

32.4 Object Attributes

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

Since the object class and type of an object can't be changed anymore once an object has been created there are only functions for querying these attributes:

```
int fl_get_object_objclass(FL_OBJECT *obj);
int fl_get_object_type(FL_OBJECT *obj);
```

Receiving a negative value indicates that a NULL pointer was passed to the functions.

To set the two colors that influence the appearance of the object use

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

and to find out about the colors of an object use

```
void fl_get_object_color(FL_OBJECT *obj,
                         FL_COLOR *col1, FL_COLOR *col2);
void fl_set_object_boxtype(FL_OBJECT *obj, int boxtype);
```

Changes the shape of the box of the object. Please note that not all possible boxtypes are suitable for all types of objects, see the documentation for the different objects for limitations.

To find out the current boxtype of an object use

```
int fl_get_object_boxtype(FL_OBJECT *obj);
```

Receiving a negative value indicates that a `NULL` pointer was passed to the function.

There are also functions to change or query the border width of an object:

```
void fl_set_object_bw(FL_OBJECT *obj, int bw);
void fl_get_object_bw(FL_OBJECT *obj, int *bw);
```

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

To change or inquire the objects position (relative to the form it belongs to) the functions

```
void fl_set_object_position(FL_OBJECT *obj, FL_Coord x, FL_Coord y);
void fl_get_object_position(FL_OBJECT *obj, FL_Coord *x, FL_Coord *y);
```

exist. If the object is visible it's redrawn at the new position.

An object can also be moved relative to its current position using the function

```
void fl_move_object(FL_OBJECT *obj, FL_Coord dx, FL_Coord dy);
```

where `dx` and `dy` are the amounts by which the object is moved to the right and down.

To change or inquire about the size of an object use

```
void fl_set_object_size(FL_OBJECT *obj, FL_Coord w, FL_Coord h);
void fl_get_object_size(FL_OBJECT *obj, FL_Coord *w, FL_Coord *h);
```

When changing the size of the object the position of its upper left hand corner remains unchanged.

To set or query both the position and the size of an object the functions

```
void fl_set_object_geometry(FL_OBJECT *obj, FL_Coord x, FL_Coord y,
                            FL_Coord w, FL_Coord h);
void fl_get_object_geometry(FL_OBJECT *obj, FL_Coord *x, FL_Coord *y,
                            FL_Coord (*w, FL_Coord *h);
```

can be used.

Please note: always use one of the above functions to change the position and/or size of an object and don't try to change the information stored in the object directly. There's some double bookkeeping going on under the hood that makes sure that the objects position and size won't change due to rounding errors when the whole form gets resized and changing the internal information kept in the objects structure would interfere with this.

There's a second function for calculation an objects geometry:

```
void fl_get_object_bbox(FL_OBJECT *obj, FL_Coord *x, FL_Coord *y,
```

```
                                FL_Coord *w, FL_Coord *h);
```

The difference between this functions and [fl_get_object_geometry()], page 287 is that [fl_get_object_bbox()], page 287 returns the bounding box size that has the label, which could be drawn outside of the object figured in.

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

```
FL_OBJECT *fl_get_object_component(FL_OBJECT *obj, int objclass,
                                   int type, int number);
```

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

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

To influence change the color, font size, font style, alignment and text of the label of an object use

```
void fl_set_object_lcolor(FL_OBJECT *obj, FL_COLOR 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);
void fl_set_object_label(FL_OBJECT *obj, const char *fmt, ...);
```

To find out about the object labels color, font size, style, alignment and the string itself use

```
FL_COLOR fl_get_object_lcolor(FL_OBJECT *obj);
int fl_get_object_lsize(FL_OBJECT *obj);
int fl_get_object_lstyle(FL_OBJECT *obj);
int fl_get_object_lalign(FL_OBJECT *obj);
const char * fl_get_object_label(FL_OBJECT *obj);
```

To set a tool-tip text for an object use the following routines

```
void fl_set_object_helper(FL_OBJECT *obj, const char *helpmsg);
void fl_set_object_helper_f(FL_OBJECT *obj, const char *fmt, ...);
```

where helpmsg is a text string (with possible embedded newlines in it) that will be shown when the mouse hovers over the object for nore than about 600 msec. A copy of the string is made internally. The second functions accepts instead of a simple string a format string just as it's used for printf() etc., followed by as many further arguments as the format string contains format specifiers.

The boxtype, color and font for the tool-tip message displayed can be customized further using the following routines:

```
void fl_set_tooltip_boxtype(int boxtype);
void fl_set_tooltip_color(FL_COLOR textcolor, FL_COLOR background);
```

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

where boxtype is the backface of the form that displays the text. The default is [FL_BORDER_BOX], page 109. textcolor and background specify the color of the text and the color of the backface. The defaults for these are FL_BLACK and FL_YELLOW. style and size are the font style and size of the text.

There are four function for controlling how an object reacts to resizing the form it belongs to or to find out what its current settings are:

```
void fl_set_object_resize(FL_OBJECT *obj, unsigned int howresize);
void fl_get_object_resize(FL_OBJECT *obj, unsigned int *howresize);
void fl_set_object_gravity(FL_OBJECT *obj, unsigned int NWgravity,
                           unsigned int SEgravity);
void fl_get_object_gravity(FL_OBJECT *obj, unsigned int *NWgravity,
                           unsigned int *SEgravity);
```

See Chapter 4 [Doing Interaction], page 37, for more details on the resizing behaviour of objects.

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 of the two functions

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

The form is automatically redrawn once it is "unfrozen", so a call of [fl_redraw_form()], page 297 isn't required (and, while the form is "frozen", calling this function as well as [fl_redraw_object()], page 297 has no effects).

You may also freeze and unfreeze all forms at once by using

```
void fl_freeze_all_forms(void);
void fl_unfreeze_all_forms(void);
```

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], page 244 event, use the following routine

```
void fl_set_object_automatic(FL_OBJECT *obj, int yes_no);
```

To determine if an object receives [FL_STEP], page 244 events use

```
int fl_object_is_automatic(FL_OBJECT *obj);
```

To enable or disable an object to receive the [FL_DBLCLICK], page 243 event use the following routine

```
void fl_set_object_dblclick(FL_OBJECT *obj, unsigned long timeout);
```

where timeout specifies the maximum time interval (in msec) between two clicks for them to be considered a double-click (using 0 disables double-click detection). To determine the current setting of the timeout use

```
unsigned fl_get_object_dblclick(FL_OBJECT *obj);
```

To make an object or a group invisible or visible use the following two functions

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

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

obj can be the pseudo-object returned by [fl_bgn_group()], page 285 and then allows to hide or show whole groups of objects.

To determine if an object is visible (given that the form it belongs to is also visible) use

```
int fl_object_is_visible(FL_OBJECT *obj);
```

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

returns obj to the application program after calling its callback if one exists.

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

sets the input focus in form form to object obj. Note however, if this routine is used as a response to an [FL_UNFOCUS], page 244 event, i.e., as an attempt to override the focus assignment by the main loop from within an objects event handler, this routine will not work as the main loop assigns a new focus object upon return from the object event handler, which undoes the focus change inside the event handler. To override the [FL_UNFOCUS], page 244 event the following routine should be used:

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

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

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

The routine

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

binds a callback routine to an object.

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

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

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

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

binds a callback routine to an entire form.

It is sometimes useful to obtain the last X 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);
```

In other rare circumstances one might not be interested not in the X event but instead the internal XForms event resulting in the invocation of an object or form callback. This information can be obtained by calling

```
int fl_current_event(void);
```

A callback invocation resulting from a call of [fl_call_object_callback()], page 290 will return FL_TRIGGER. For other possible return value see Chapter 26 [the chapter about XForms internal events], page 242. Calling this function is only useful while within an object or form callback, at all other times it returns just FL_NOEVENT.

Also in objects callback it might be of interest to find out if the mouse is on top of a certain letter of the (inside) label (one trivial use of this can be found in the program `demo/strange_button.c`. To find out about this use

```
int fl_get_label_char_at_mouse(FL_OBJECT *obj);
```

The function returns the index of the character in the label of the object the mouse is on or `-1` if it's not over the label. Note that this function has some limitations: it can only be used on labels inside of the object and the label string may not contain underline characters (and the label can't be a symbol) - if you try to use it on labels that don't satisfy these requirements `-1` is returned.

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

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

The function will output the specified object in PostScript. If `fname` is `NULL`, a file selector will be shown to ask the user for a file name. The function returns a negative number if no output is generated due to errors. At the moment, only the `FL_XYPLOT` object is supported. Nothe that this function isn't part of the statndard XForms library (`libforms`) but the XForms image library (`libflimage` discussed in Chapter 37 [Part VI Images], page 319.

The object must be visible at the time of the function call. The hardcopy should mostly be WYSIWYG and centered on the printed page. The orientation is determined such that a balanced margin results, i.e., if the width of the object is larger than the height, landscape mode will be used. Further, if the object is too big to fit on the printed page, a scale factor will be applied so the object fits. The box underneath the object is by default not drawn and in the default black&white mode, all curves are drawn in black. See demo program `xyplotover.c` for an example output.

It is possible to customize the output by changing the PostScript output control parameters via the function

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

A typical use is to call this routine to obtain a handle to the PostScript output control structure and change the control structure members to suit your needs before calling `[fl_object_ps_dump()], page 291`. You should not free the returned buffer.

The control structure has the following members

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

`int orientation`
: Valid choices are `FLPS_AUTO`, `FLPS_PORTRAIT` and `FLPS_LANDSCAPE`. The default is `FLPS_AUTO`.

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

`int eps` Set this to 1 if output in EPS format is required.

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

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

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

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

To generate a PostScript output of a form or forms, use the `fd2ps` program documented in Chapter 13 [Part II Generating Hardcopies], page 104.

32.5 Doing Interaction

To display the form `form` on the screen use one of

```
Window fl_show_form(FL_FORM *form, int place, int border,
                    const char *title);
Window fl_show_form(FL_FORM *form, int place, int border,
                    const char *fmt, ...);
```

`place` controls the position and size of the form. `border` indicates whether a border (window manager's decoration) should be drawn around the form. If a border is to be drawn `title` is the name of the window (and its associated icon). The routine returns the window identifier of the form. For resource and identification purposes, the form name is taken to be the title with spaces removed and the first character lower-cased. E.g., if a form has a title `"Foo Bar` the forms name is derived as `"fooBar"`. The only difference between the two functions is that the first one accepts a simple string for the title while the second expects a format string like `printf()`, followed by the appropriate number of arguments.

For the the location and size of the window controlled 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()], page 294`. Interactive resizing is allowed.

`FL_PLACE GEOMETRY`
Place at the latest position and size (see also below) or the geometry set via `[fl_set_form_geometry()], page 294` etc. 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], page 38 as in [FL_PLACE_MOUSE], page 37|[FL_FREE_SIZE], page 38.

FL_PLACE_CENTER
: The form is placed in the center of the screen. If [FL_FREE_SIZE], page 38 is also specified, interactive resizing will be allowed.

FL_PLACE_FULLSCREEN
: The form is scaled to cover the full screen. If [FL_FREE_SIZE], page 38 is also specified, interative 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()], page 294, will be used, otherwise interactive 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], page 38 is also specified, interactive resizing will be allowed.

FL_PLACE_CENTERFREE
: Same as [FL_PLACE_CENTER], page 37|[FL_FREE_SIZE], page 38, 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.

If no size is 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.

There are three values that can be passed for border:

FL_FULLBORDER
: Draw full border with title

FL_TRANSIENT
: Draw borders with possibly less decoration (depends on the window managers behaviour)

FL_NOBORDER
: Draw no border at all

Since multiple forms can be displayed at the same time note that using 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 preferable. Also note that you can't iconify a form that has no borders and under most window managers forms displayed with `FL_TRANSIENT` can't be iconified either.

One 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 (must be called before any full-bordered form is shown):

```
void fl_set_app_nomainform(int yes_no);
```

with a true flag.

All visible forms will have the properties `WM_CLASS`, `WM_CLIENT_MACHINE` and `WM_NAME` set. In addition, the first full-bordered form will have the `WM_COMMAND` property set and is by default the applications main form.

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

```
Window fl_prepare_form_window(FL_FORM *form, int place, int border,
                              const char *name);
Window fl_prepare_form_window_f(FL_FORM *form, int place, int border,
                                const char *fmt, ...);
```

These routines create a window that obeys any and all constraints just as `[fl_show_form()], page 292` does but remains unmapped. The only difference between the two functions is that the first one takes a simple string for the forms name while the second expects a format string like `printf()`, followed by the appropriate number of further arguments.

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()], page 292`.

The application program can raise a form to the top of the screen so no other forms obscures it by calling

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

To instead lower a form to the bottom of the stack use

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

When placing a form on the screen using `FL_PLACE_GEOMETRY` for the `place` argument to `[fl_show_form()], page 292` the position and size can be set before by 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_set_form_geometry(FL_FORM form*, FL_Coord x, FL_Coord y,
                          FL_Coord w, FL_Coord h);
void fl_scale_form(FL_FORM *form, double xsc, double ysc);
```

where [fl_set_form_geometry()], page 294 combines the functionality of [fl_set_form_position()], page 294 and [fl_set_form_size()], page 294 and the last routine, [fl_scale_form()], page 294, scales the form in horizontal and vertical direction by the factors passed to the function. These routines can also be used when the form is visible.

Sometimes it is desirable to know how large the decoration are the window manager puts around a forms window. They can be obtained by a call of

```
void fl_get_decoration_sizes(FL_FORM *form, int *top, int *right,
                             int *bottom, int *left);
```

This is especially useful if it is necessary to open a window at some previously stored position since in that case one needs the position of of the window, which deviates from the position reported for the form by the window manager's decorations. Obviously, the above function can't be used for forms that are embedded into another form.

The function

```
int fl_form_is_iconified(FL_FORM *form);
```

allows to test if the (visible) window of a form is in iconified state.

If interactive resizing is allowed (e.g., by showing the form with [FL_PLACE_POSITION], page 37) it can be useful to limit the range of the size of a form can take. To this end, the following functions are available

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

Although these two routines can be used before or after a form becomes visible, not all window managers honor such requests once the window is visible. Also note that the constraints only apply to the next call of [fl_show_form()], page 292 for the form.

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

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

where icon can be any valid pixmap ID. (see Section 15.6 [Pixmap Object], page 114, for some of the routines that can be used to create pixmaps.) Note that a previously set icon if not freed or modified in anyway.

If, for any reason, you would like to change the form title after the form has been made visible, the following calls can be used (they will also change the icon title)

```
void fl_set_form_title(FL_FORM *form, const char *name);
void fl_set_form_title_f(FL_FORM *form, const char *fmt, ...);
```

(While the first function expects a simple string, the second has to be called with a format string as printf() etc., followed by the corresponding number of arguments.)

The routine

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

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

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

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

The function can return that the form is visible (`[FL_VISIBLE], page 42`), is invisible (`[FL_INVISIBLE], page 42`) or is in the processing of becoming invisible (`[FL_BEING_HIDDEN], page 42`).

The most important function for doing the actual interaction with forms is

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

It starts the main loop of the program and returns only when either the state of an object changes that has no callback bound to it or `[fl_finish()], page 285` is called in a callback. In the first case the address of the object is returned, in the latter `NULL`.

A second way of doing interaction with the currently displayed forms is using

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

This routine returns `NULL` immediately unless the state of one of the object (without a callback bound to it) changed. In that case a pointer to this object gets returned. `NULL` also gets returned after a call of `[fl_finish()], page 285`.

Then there are two more functions:

```
FL_OBJECT *fl_do_only_forms(void);
FL_OBJECT *fl_check_only_forms(void);
```

Both functions do the same as `[fl_do_forms()], page 296` and `[fl_check_forms()], page 296` except that they do not handle user events generated by application windows opened via `[fl_winopen()], page 303` or similar routines.

To activate or deactivate a form for user interaction you can use

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

The same can also be done for all forms at once using

```
void fl_deactivate_all_forms(void)
void fl_activate_all_forms(void)
```

To find out if a form is currently active call

```
int fl_form_is_activated(FL_FORM *form);
```

A return value of 0 tells you that the form is currently deactivated.

You can also register callbacks for a form 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);

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

Also individual objects (or groups of objects if the argument of the function is an object returned by [fl_bgn_group()], page 285) can be activated and deactivated to enable or disable user interaction:

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

It is normally useful to give the user a visual clue when an object gets deactivated, e.g., by graying out its label etc.

To find out if an object is active use

```
int fl_object_is_active(FL_OBJECT *obj);
```

```
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 objects when necessary, but there might be situations in which an explicit redraw is required.

To redraw an entire form use

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

For non-form windows, i.e., those created with [fl_winopen()], page 303 or similar routines by the application program, the following means of interaction are provided (note that these do not work on form windows, for which a different set of functions exist, see Section 33.2 [Windowing Support], page 302, for details.)

You may set up a callback routine (of type FL_APPEVENT_CB for all user events using

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

The function returns the previously set callback (or NULL).

It is also 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 the event callback functions for the window specified, which will be called when an event of type xevent_type is pending for the window. If xevent_type is 0 it signifies a callback for all event for window win. Note that the Forms Library does not solicit any event for the caller, i.e., the Forms Library assumes the caller opens the window and solicits all events before calling these routines.

To let the Forms Library handle event solicitation, the following function may be used

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

32.6 Signals

Typically, when a signal is delivered, the application does not know what state the application is in, thus limiting the tasks a signal handler can do. In a GUI system and with a main loop inside the library, it's even harder to know what's safe or unsafe to do in a signal handler. Given all these difficulties, the Forms Library's main loop is made to be aware of signal activities and invoke signal handlers only when it's appropriate to do so, thus removing most limitations on what a signal handler can do.

The application program can elect to handle the receipt of a signal by registering a callback function that gets called when a signal is 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()], page 298 will store the callback function and initiate a 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 of XForms's functions as well as Xlib functions as if they were reentrant. Further, a signal callback registered his way 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 signals, a code fragment similar to the following can be used:

```
void clean_up(int signum, void *data) {
    /* clean up, of course */
}

/* call this somewhere after fl_initialize() */
fl_add_signal_callback(SIGINT, clean_up, &mydata);
```

After this, whenever a `SIGINT` signal is received, `clean_up()` is called.

To remove a signal callback, the following routine should be used

```
void fl_remove_signal_callback(int signal);
```

Although very easy to use, 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()], page 298 prevents the application program from using any, potentially more flexible, system signal handling routines on some platforms. Also there might be perceptible delays from the time a signal is delivered by the OS and the time its callback is invoked by XForms' main loop. This delay can be particular troublesome for timing sensitive tasks (playing music for example).

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()` for example).

To change the default behavior of the built-in signal facilities, the following routine should be called prior to any use of `fl_add_signal_callback()` with a true value for `flag`:

```
void fl_app_signal_direct(int flag);
```

After this call [fl_add_signal_callback()], page 298 will not initiate any actions to receive a signal. The application program should handle the receipt and blocking of signals (via e.g., 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, possibly after performing some timing sensitive tasks:

```
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 invocations of [fl_signal_caught()], page 299 occur before the main loop is able to call the registered callback, the callback is called only once.

The following example illustrates how to handle a timing critical situation (for most application, idle callback, timeouts or FL_TIMER object should be sufficient).

First, you need to define the function that will handle the timing critical tasks. The function will be registered with the OS to be invoked directly by it. There are limitations on what you can do within a (OS) signal handler, in particular, GUI activity is not safe.

```
void timing_critical_task(int sig) {
    /* handle timing critical tasks that does not involve GUI */
    ...
    /* Now tell the library the signal has been delivered by the OS.
     * The library will invoke the xforms signal handler when it's
     * appropriate to do so */
    fl_signal_caught(sig);
}
```

Now define a (XForms) signal handler that will be responsible for handling the response of the GUI upon receipt of the signal

```
void gui_signal_handler(int sig, void *data) {
    /* within an XForms signal handler, there is no limitation
     * on GUI activitity */
    fl_set_object_color(....);
    ...
}
```

To make all this work, a set-up similar to the following can be used

```
/* setup the signal */
fl_app_signal_direct(1);
setitimer(ITIMER_REAL, interval);

/* setup the OS signal handler */
signal(SIGALRM, timing_critical_tasks);

/* setup the XForms signal handler */
fl_add_signal_callback(SIGALRM, gui_signal_handler, &myData);
```

32.7 Idle Callbacks and Timeouts

For application programs that need to perform some light, but semi-continuous or periodic tasks, idle callback and timeouts (also FL_TIMER objects) can be utilized.

To register an idle callback with the system, use the following routine

```
typedef int (*FL_APPEVENT_CB)(XEvent *, void *);
FL_APPEVENT_CB fl_set_idle_callback(FL_APPEVENT_CB callback,
                                    void *user_data);
```

where `callback` is the function that will get called whenever the main loop is idle. The time interval between invocations of the idle callback can vary considerably depending on interface activity and other factors. A range between 50 and 300 msec should be expected. While the idle callback is executed it won't be called again (i.e., no call of any XForms function from within the idle callback function will call the idle callback function), so it does not need to be reentrant.

It is possible to change what the library considers to be "idle" with the following function:

```
void fl_set_idle_delta(long msec);
```

Here `msec` is the minimum time interval of inactivity after which the main loop is considered to be in an idle state. However it should be noted that under some conditions an idle callback can be called sooner than the minimum interval.

If the timing of the idle callback is of concern, timeouts should be used. Timeouts are similar to idle callbacks but with the property that the user can specify a minimum time interval that must elapse before the callback is called. The precision of timeouts tends to be quite a bit better than that of idle callbacks since they internally get prefered treatent. To register a timeout callback, the following routine can be used

```
typedef void (*FL_TIMEOUT_CALLBACK)(int, void *);
int fl_add_timeout(long msec, FL_TIMEOUT_CALLBACK callback,
                   void *data);
```

The function returns the timeout ID (note: the function will not return 0 and -1, so the application can use these values to mark invalid or expired timeouts). When the time interval specified by the `msec` argument (in milli-second) is elapsed, the timeout is removed and the callback function is called with the timeout ID as the first argument. Although a timeout offers some control over the timing, due to performance and CPU load compromises, while the resolution can be better than 10 ms under favourable conditions, it can also be much worse, occasionally up to 150 ms.

To remove a timeout before it triggers, use the following routine

```
void fl_remove_timeout(int id);
```

where `id` is the timeout ID returned by `[fl_add_timeout()], page 300`. See Section 21.1 [Timer Object], page 186, for the usage of `FL_TIMER` object. For tasks that need more accurate timing the use of signal should be considered.

32.8 Global Variables

For convenience the library exports a number of global variables. These are:

`FL_OBJECT *FL_EVENT`

This is a special object returned by `[fl_do_forms()], page 296` etc. when an X event is received that isn't coming from a form under the control of the library, e.g., for a window that was opened directly via Xlib functions. Upon receiving this special event the application program can and must remove the pending event from the queue using `[fl_XNextEvent()], page 49`.

`FL_FORM *fl_current_form`
: This variable is always set to the currently active form.

`Display *fl_display`
: This variable is set to the display (X server) the program is connected to and is needed as an argument for many Xlib functions. It's recommended not to use this global variable but instead either the function `[fl_get_display()], page 255` or `[FL_FormDisplay()], page 255` (the latter accepts a form pointer as its argument and will also be safe in future versions of the library that may support multiple connections).

`int fl_screen`
: This variable is set to the default screen of the display connection.

`Window fl_root`
: This variable is set to the root window.

`Window fl_vroot`
: Some window managers have problems with obtaining the corrent root window and applications don't work with the normal root windows. In this case `fl_vroot` can be used instead.

`int fl_scrw, fl_scrh`
: These variables contain the screens width and height.

`int fl_mode`
: The variable contains the visual mode in use, it should be one of the Xlib constants `PseudoColor`, `TrueColor`, `DirectColor`, `StaticColor`, `GrayScale` or `StaticGray. Alternatively, the functions [fl_get_vclass()], page 254 or fl_get_form_vclass()` can be used (the latter accepts a form pointer as its argument and is thus also safe for future versions that may allow multiple connections).

`FL_State fl_state[ 6 ]`
: This array of structure of type `[FL_State], page 254` contains a lot of information about the graphics mode, where each structure has the information for each of the cisual modes. Interesting is only the entry for the visual mode used, `[fl_vmode], page 301`.

`char *fl_ul_magic_char`
: This variable points to the character used to indicate underlining in labels and other texts. If it appears as the very first character of a string all characters in that string are underlined, otherwise the character direct in front of it. Per default it's set to `'\b'`.

33 Some Useful Functions

33.1 Misc. Functions

The following routine can be used to sound the keyboard bell (if capable):

```
void fl_ringbell(int percent);
```

where `percent` can range from -100 to 100 with 0 being the default volume setting of the keyboard. A value of 100 indicates maximum volume and a value of -100 minimum volume (off). Note that not all keyboards support volume variations.

To get the user name who's running the application you can use the routine

```
const char *fl_whoami(void);
```

To get a string form of the current date and time, the following routine is available:

```
const char *fl_now(void);
```

The format of the string is of the form `"Wed Jun 30 21:49:08 1993"`.

The following time related routine might come in handy

```
void fl_gettime(unsigned long *sec, unsigned long *usec);
```

Upon function return `sec` and `usec` are set to the current time, expressed in seconds and microseconds since 00:00 GMT January, 1970. This function is most useful for computing time differences.

Th function

```
int fl_mode_capable(int mode, int warn);
```

allows to determine the visual classes the system is capable of. `mode` must be one of `GrayScale`, `StaticGray`, `PseudoColor`, `StaticColor`, `DirectColor` and `TrueColor` and the function returns 1 if the system is capable of displaying in this visual class and 0 otherwise. If `warn` is set a warning is printed out in case the capability asked for isn't available.

Finally

```
int fl_msleep(usigned long msec);
```

allows to wait for a number of milli-seconds (with the best resolution possible on your system).

33.2 Windowing Support

Some of the following routines are also 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 be subject to changes, both in their API and/or functionality.

You can create and show a window with the following routines

```
Window fl_wincreate(const char *name);
Window fl_winshow(Window win);
```

where the parameter `win` of `[fl_winshow()], page 302` is the window ID returned by `[fl_wincreate()], page 302`. The title of the window is set by the `name` argument.

Between the creation and showing of the window other attributes of the window can be set. Note that a window opened this way is always a top level window and uses all the Forms

Library's defaults (visual, depth etc.). Another thing about [fl_winshow()], page 302 is that it will wait for and gobble up the first Expose event and you can draw into the window immediately after the function returns.

It is sometimes more convenient to create and show a window in a single call using

```
Window fl_winopen(const char *name);
```

This will open a (top-level) window with the title name. A window so opened can be drawn into as soon as the function returns, i.e., [fl_winopen()], page 303 waits until the window is ready to be drawn to.

The newly opened window will have the following default attributes

event_mask
: ExposureMask, KeyPressMask, KeyReleaseMask, ButtonPressMask, ButtonReleaseMask, OwnerGrabButtonMask, ButtonMotionMask, PointerMotionMask, PointerMotionHintMask, StructureNotifyMask

backing_store
: as set by fl_cntl.backingStore

class
: InputOutput

visual
: same as Forms Library's default

colormap
: same as Forms Library's default

To make a top-level window a sub-window of another window use the following routine

```
int fl_winreparent(Window win, Window new_parent);
```

The origin of the window win will be at the origin of the parent window new_parent. At the time of the function call, both the window and the parent window must be valid windows.

By default, a newly opened window will have a size of 320 by 200 pixels and no other constraints. You can modify the default or constraints using the following routines prior to calling [fl_winopen()], page 303:

```
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()], page 303 in addition will make the window non-resizeable (but 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).

Also the state of the window when opening it can be influenced by the function

```
void fl_initial_winstate(int state);
```

where state is on of the XLib constants NormalState (the default) or IconicState, which will result in the opened window being iconified. The third possible constant, WithdrawnState, doesn't make much sense in this context.

It is sometimes desirable to have a window that is resizeable but only within a useful range. To set such a constraint use the following functions:

```
void fl_winminsize(Window window, FL_Coord minw, FL_Coord minh);
```

```
void fl_winmaxsize(Window window, FL_Coord maxw, FL_Coord maxh);
```

These two routines can also be used after a window has become visible. For windows still to be created/opened, use `None` 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(None, 100,100);
fl_winmaxsize(None, 1024,768)
win = fl_winopen("MyWin");
```

In addition to the window size preference you can also set the 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.

Alternatively, you can set the geometry (position and size) in a single 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, windows for which `[fl_wingeometry()], page 304` had been created will not allow interactive resizing later on.

There are further routines that can be used to change other aspects of the window to be created:

```
void fl_winaspect(Window win, FL_Coord x, FL_Coord y);
```

This will set the aspect ratio of the window for later interactive resizing.

To change the window title (and its associated icon title) use

```
void fl_wintitle(Window win, const char *title);
void fl_wintitle_f(Window win, const char *fmt, ...);
```

While the first function only accepts a simple string for the window title the second one allows to pass a format string just like the one used for `printf()` etc. and an appropriate number of further arguments which are used to construct the title.

To change the icon title only use the routines

```
void fl_winicontitle(Window win, const char *title);
void fl_winicontitle_f(Window win, const char *fmt, ...);
```

To install an icon for the window use

```
void fl_winicon(Window win, Pixmap icon, Pixmap mask);
```

You can suppress the window manager's decoration or make a window a transient one by using the following routines prior to creating the window

```
void fl_noborder(void);
void fl_transient(void);
```

You can also set the background of the window to a certain color using the following call

```
void fl_winbackground(Window win, unsigned long pixel);
```

It is possible to set the steps by which the size of a window can be changed by using

```
void fl_winstepsize(Window win, int xunit, int yunit);
```

where `xunit` and `yunit` are the number of pixels of changes per unit in x- and y- directions, respectively. Changes to the window size will be multiples of these units after this call. Note that this only applies to interactive resizing.

To change constraints (size and aspect ratio) on an active window, you can use the following routine

```
void fl_reset_winconstraints(Window win);
```

The following routines are available to get information about an active window win:

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

All values returned are in pixels. The origin of a window is measured from the upper left hand corner of the root window.

To change the size of a window programmatically the following function is available:

```
void fl_winresize(Window win, FL_Coord neww, FL_Coord newh);
```

Resizing will not change the origin of the window (relative to the root window). While the window gets resized originally set restraints will remain unchanged. E.g., if a window was not permitted to be resized interactively it will continue to remain unresizeable by the user.

To move a window without resizing it use the following function:

```
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);
```

The following routine is available to iconify a window

```
int fl_iconify(Window win);
```

The return value is nonzero when the message, asking for iconification of the window, was send successfully to the window manager, otherwise zero (but this may not be taken as a sure sign that the window was really iconified).

To make a window invisible use

```
void fl_winhide(Window win);
```

A window hidden this way can be shown again later using `[fl_winshow()], page 302`.

To hide and destroy a window, use the following calls

```
void fl_winclose(Window win);
```

There will be no events generated from `[fl_winclose()], page 305`, i.e., the function waits and gobbles up 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 ID is valid or not

```
int fl_winisvalid(Window win);
```

Note that excessive use of this function may negatively impact performance.

Usually an X application should work with window managers and accepts the keyboard focus assignment. In some special situations, explicit override of the keyboard focus might be warranted. To this end, the following routine exists:

```
void fl_winfocus(Window win);
```

After this call keyboard input is directed to window `win`.

33.3 Cursors

XForms provides a convenience function to change the cursor shapes:

```
void fl_set_cursor(Window win, int name);
```

where `win` must be a valid window identifier and `name` is one of the symbolic cursor names (shapes) defined by standard X or the integer values returned by `[fl_create_bitmap_cursor()], page 306` or one of the Forms Library's pre-defined symbolic names.

The X standard symbolic cursor names (all starts with `XC_`) are defined in `<X11/cursorfont.h>` (you don't need to explicitly include this as `<forms.h>` already does this for you). For example, to set a watch-shaped cursor for form `form` (after the form is shown), the following call may be made

```
fl_set_cursor(form->window, XC_watch);
```

The Forms Library defines a special symbolic constants, `FL_INVISIBLE_CURSOR` that can be used to hide the cursor for window `win`:

```
fl_set_cursor(win, FL_INVISIBLE_CURSOR);
```

Depending on the structure of the application program, a call of `XFlush(fl_get_display());` may be required following `[fl_set_cursor()], page 306`.

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 with

```
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 define 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 calls of `[fl_set_cursor()], page 306` and `[fl_set_cursor_color()], page 306` etc.

Finally, there is a routine to create animated cursors where several cursors are displayed one after another:

```
int fl_create_animated_cursor(int *cur_names, int interval);
```

The function returns the cursor name (ID) that can be shown later via [fl_set_cursor()], page 306. In the function call cur_names is an array of cursor names (either X standard cursors or cursor names returned by [fl_create_bitmap_cursor()], page 306), terminated by -1. Parameter interval indicates the time each cursor is displayed before it is replaced by the next in the array. An interval about 150 msec is a good value for typical uses. Note that there is currently a limit of 24 cursors per animation sequence.

Internally animated cursor works by utilizing the timeout callback. This means that if the application blocks (thus the main loop has no chance of servicing the timeouts), the animation will stop.

See demo program cursor.c for an example use of the cursor routines.

33.4 Clipboard

Clipboard is implemented in the Forms Library using the X selection mechanism, more specifically the XA_PRIMARY selection. X selection is a general and flexible way of sharing arbitrary data among applications on the same server (the applications are of course not necessarily running on the same machine). The basic (and over-simplified) concept of the X selection can be summarized as follows: the X Server is the central point of the selection mechanism and all applications running on the server communicate with other applications through the server. The X selection is asynchronous in nature. Every selection has an owner (an application represented by a window) and every application can become owner of the selection or lose the ownership.

The clipboard in Forms Library is a lot simpler than the full-fledged X selection mechanism. The simplicity is achieved by hiding and handling some of the details and events that are of no interests to the application program. In general terms, you can think of a clipboard as a read-write buffer shared by all applications running on the server. The major functionality you want with a clipboard is the ability to post data onto the clipboard and request the content of the clipboard.

To post data onto the clipboard, use the following routine

```
typedef int (*FL_LOSE_SELECTION_CB)(FL_OBJECT *obj, long type);

int fl_stuff_clipboard(FL_OBJECT *obj, long type,
                       const void *data, long size,
                       FL_LOSE_SELECTION_CB callback);
```

where size is the size (in bytes) of the content pointed to by data. If successful, the function returns a positive value and the data will have been copied onto the clipboard. The callback is the function that will be called when another application takes ownership of the clipboard. For textual content the application that loses the clipboard should typically undo the visual cues about the selection. If no action is required when losing the ownership a NULLq callback can be passed. The obj argument is used to obtain the window (owner) of the selection. type is currently unused. At the moment the return value of lose_selection_callback() is also unused. The data posted onto the clipboard are available to all applications that manipulate XA_PRIMARY, such as xterm etc.

To request the current clipboard content use the following routine

```
typedef int (*FL_SELECTION_CB)(FL_OBJECT *obj, long type,
```

```
                                    const void * data, long size);

int fl_request_clipboard(FL_OBJECT *obj, long type,
                         FL_SELECTION_CB callback);
```

where `callback` is the callback function that gets called when the clipboard content is obtained. The content `data` passed to the callback function should not be modified.

One thing to remember is that the operation of the clipboard is asynchronous. Requesting the content of the clipboard merely asks the owner of the content for it and you will not have the content immediately (unless the asking object happens to own the selection). XForms main event loop takes care of the communication between the requesting object and the owner of the clipboard and breaks up and re-assembles the content if it exceeds the maximum protocol request size (which has a guaranteed minimum of 16 kB, but typically is larger). If the content of the clipboard is successfully obtained the main loop invokes the lose selection callback of the prior owner and then the requesting object's callback function.

The function returns a positive number if the requesting object owns the selection (i.e., the callback could beinvoked before the function returned) and 0 otherwise.

If there is no selection the selection callback is called with an empty buffer and the length of the buffer is set to 0. In that case `[fl_request_clipboard()], page 307` returns -1.

34 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 the 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 a somewhat 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.

34.1 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` passed to `[fl_initialize()], page 277` is a table of structures listing your applications 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()` 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()], page 277` with the following routine

```
void 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

`FL_NONE` for variables not to be used (or not available)

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()], page 309` simply looks up all entries specified in the `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` command line option.

Summarized below are the currently recognized Forms Library built-in resources:

Resource Name	**Class**	**Type**	**Default values**
rgamma	Gamma	float	1.0
ggamma	Gamma	float	1.0
bgamma	Gamma	float	1.0
visual	Visual	string	best
depth	Depth	int	best
doubleBuffer	DoubleBuffer	bool	true
privateColormap	PrivateColormap	bool	false
standardColormap	StandardColormap	bool	false
sharedColormap	SharedColormap	bool	false
pupFontSize	PupFontSize	int	12pt
buttonFontSize	FontSize	int	10pt
sliderFontSize	FontSize	int	10pt
inputFontSize	FontSize	int	10pt
browserFontSize	FontSize	int	10pt
menuFontSize	FontSize	int	10pt
choiceFontSize	FontSize	int	10pt
ulPropWidth	ULPropWidth	bool	true
ulThickness	ULThickness	int	1
scrollbarType	ScrollbarType	string	thin
coordUnit	CoordUnit	string	pixel
borderWidth	BorderWidth	int	1

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`
- ommand line options

All options set via resources may not be the final values used because resource settings are applied at the time an 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()], page 120`). Thus altering the size after the button is created via `[fl_set_object_lsize()], page 288` overrides whatever is set by the resource database.

To run your application in `PseudoColor` with a depth of 8 and a thicker underline, specify the following resources

```
appname*visual:      PseudoColor
appname*depth:       8
appname*ulThickness: 2
```

Since resources on a form by form basis are yet to be implemented, there is no point specifying anything more specific although also `appname.XForm.depth` etc. would work correctly.

34.1.1 Resources Example

Let us assume that you have an application named `myapp` and it accepts the options `-foo` *level* and `-bar` plus 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 variables `foolevel` and `ifbar` are set either through resource files or command line options, with the command line options overriding those set in the resource files.

In case neither the command line nor the resource files specified the options, the default value string is converted.

There is another routine, a resource routine of the lowest level in XForms, which might be useful if a quick-and-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` must be complete resource specifications (minus the application name) and should not contain wildcards of any kind. The resource will be converted according to the type and result stored in `type`, which is an integer of type `[FL_RTYPE], page 309`. `nbytes` is used only if the resource type is `[FL_STRING], page 310`. The function returns the string representation of the resource value. If a value of `[FL_NONE], page 310` is passed for `type` the resource is not converted and the pointer `val` is not dereferenced.

There is also a routine that allows the application program to set resources programmatically:

```
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()], page 312` and `[fl_get_resource()], page 312` can be used to store and retrieve arbitrary strings and values and may be useful to pass data around.

34.2 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 similar 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, ...);
```

Since eventually 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 the first letter lower-cased, i.e., if a form is shown with a label `Foo Bar`, the name of the form should be `fooBar`.

35 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, or they rely too much on the underlying window systems. Thus whenever possible, try not to use these routines.

35.1 Interaction

35.1.1 Form Events

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 the event can even be consumed before the libraries 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 the 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`
- `FL ALL EVENT` (see below)

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 possible to register a handler only for one of them (but 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()], page 53`.

Once an event handler is registered and the event is detected, then instead of doing the default processing by the dispatcher, the registered handler function is invoked. The handler function must return either `FL_PREEMPT` if the event is consumed) and `0` otherwise so that the internal processing of the event can continue. See the demo program `minput2.c` for an example.

Since these kind of handlers work on a rather low level there's a chance that they interfere with some mechanisms of the library. Consider the case of setting a raw callback handler

for mouse press and release events, in which the handler returns `0` for mouse press events but `FL_PREEMPT` on relese events. In that case the mouse press event results in the normal processing and e.g., a button below the mouse will receive it (and be drawn correspondingly). To be drawn again in its normal way it also needs to receive the release event (even if the mouse isn't on top of it anymore when the mouse button is released). But when the handler function doesn't also let the release event propagate to the normal handling of events then the button will never receive the expected release event and will stay drawn in the way as if the release event never happened. Thus one should avoid having different return values from the handler for pairs of related events.

35.1.2 Object Events

Just as you can by-pass the internal event processing for a particular form, you can also do so for an object. Unlike in raw callbacks, you can not select individual events.

The mechanism 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.

In Chapter 31 [the chapter about pre-emptive handlers], page 275, the API was already discussed in detail, so here we just repeat the discussion for completeness as any use of pre-emptive handler is considered "dirty tricks".

To register a pre-handler, use the following routine

```
typedef int (*FL_HANDLEPTR)(FL_OBJECT *obj, 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. The arguments `mx` and `my` are the mouse position and `key` is the key pressed. The last parameter, `raw_event` is a pointer to the XEvent that caused the invocation of the pre-handler. cast to a void pointer.

Notice that the pre-handler has the same function prototype as the built-in handler. Actually they are called with the exact same parameters by the event dispatcher. The prehandler should return `0` 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.

See demo program `preemptive.c` for an example.

A similar mechanism exists for registering a post-handler, i.e., a handler invoked after the built-in handler is finished, by using

```
void fl_set_object_posthandler(FL_OBJECT *, FL_HANDLEPTR prehandler);
```

Whenever possible a post-handler should be used instead of a pre-handler.

35.2 Other

As stated earlier, `[fl_set_defaults()], page 279` can be used to modify the Forms Library's defaults prior to calling `[fl_initialize()], page 277`. Actually, this routine can also be used after `[fl_initialize()], page 277` to override the values set on the command line or in the application databases. However, overriding users' preferences should

be done with discretion. Further, setting `privateColormap` after `[fl_initialize()]`, `page 277` has no effect.

36 Trouble Shooting

This chapter 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()`], page 292 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 `fl_update_display(1)` 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()`], page 54.

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 execution or a long task without involving the main loop of Forms Library. You can force a screen update using `fl_update_display(1)`.

I found a bug in XForms, What do I do?

Please consider subscribing to the XForms mailing list at

`http://lists.nongnu.org/mailman/listinfo/xforms-development`

and sending an email with information about the bug you found. Please try to post information about the version of the Forms Library you're using and your OS beside a description of the bug. Some sample code that exhibits the erratic behavior would help greatly.

If, for some reasons, you don't want subscribe to the mailing list you may also send an email to one of the maintainers. At the moment you probably should first contact Jens Thoms Toerring, `<jt@toerring.de>`.

Part VI - Image Support API

37 Images

Although images are not typically a part of the GUI, they are often part of an application. For this reason and others, image support is part of Forms Library. It is somewhat not unexpected that the users of a graphical user interface want some graphics support.

The most important reason to have image support in the library is the amount of questions/requests on the mailing list of the Forms Library about images. It convinced us that having image support will make many Forms Library users life easier.

The second reason has something to do with image support in X, which at best is cumbersome to use as the API reflects the underlying hardware, which, at the level of Xlib, is quite appropriate, but not quite what an application programmer wants to deal with. Image support in Forms Library for the large part is hardware independent. This is possible because xforms makes distinction between the real image it keeps and the image being displayed. At the expense of some flexibility and memory requirement, the high-level image support API should prove to be useful for most situations.

The third reason is that image support as it is now in the library is well isolated and is only linked into an application when it is actually being used. This is not a trivial point in the consideration to include image support in the library proper.

37.1 The Basic Image Support API

Reading and displaying images are quite easy. It can be as simple as a couple of lines of code:

```
FL_IMAGE *image;

if ((image = flimage_load("imagefilename"))
    image->display(image, win);
```

In this example, an image is created from a file, then the image is displayed in a window, `win`. For most casual uses, this is really what is needed to load and display an image.

As you may have guessed, an image in Forms Library is represented by a structure of type `FL_IMAGE`. In addition to the pixels in the image, it also keeps a variety of information about the image such as its type, dimension, lookup tables etc. Further, if the image can not be displayed directly on the display hardware (for example, the image is 24 bits, while the display is only capable of 8 bits), a separate displayable image is created and displayed. Any manipulation of the image is always performed on the original high-resolution image, and a new displayable image will be created if necessary.

Writing an image is just as simple

```
if (flimage_dump(image, "filename", "jpeg") < 0)
    fprintf(stderr,"image write failed");
```

In this code snippet, an image in memory is written to a file in JPEG format. As you might have noticed by now, all image routines start with flimage. The exact APIs for reading and writing an image are as follows

```
FL_IMAGE *flimage_load(const char *filename);
int flimage_dump(FL_IMAGE *im, const char *filename, const char *fmt);
```

The function [flimage_load()], page 319 takes a filename and attempts to read it. If successful, an image (or multiple images) is created and returned. If for any reason the image can't be created (no permission to read, unknown file format, out of memory etc), a null pointer is returned. As will be documented later, error reporting and progress report can be configured so these tasks are performed inside the library.

The function [flimage_dump()], page 319 takes an image, either returned by [flimage_load()], page 319 (possibly after some processing) or created on the fly by the application, attempts to create a file to store the image. The image format written is controlled by the third parameter fmtq, which should be either the formal name or the short name of one of the supported formats (such as jpeg, ppm, gif, bmp etc., see section 23.3) or some other formats the application knows how to write. If this parameter is NULL, the original format the image was in is used. If the image is successfully written, a non-negative number is returned, otherwise a negative number. Depending on how the image support is configured, error reporting may have already occurred before the function returns.

Given these two routines, a file converter (i.e., changing the image file format) is simple

```
if ((image = flimage_load("inputfile"))
    flimage_dump(image, "outfile", "newformat");
```

See the demo program iconvert.c for a flexible and usable image converter.

To free an image, use the following routine

```
void flimage_free(FL_IMAGE *image);
```

The function first frees all memory allocated for the image, then the image structure itself. After the function returns, the image should not be referenced.

The following routines are available to display an image in a window

```
int flimage_display(FL_IMAGE *image, FL_WINDOW win);
int flimage_sdisplay(FL_IMAGE *image, FL_WINDOW win);
```

where win is a window ID. If the image(s) is successfully displayed, a non-negative integer is returned, a negative integer otherwise. The difference between the two display routines is that [flimage_sdisplay()], page 320 only displays a single image while [flimage_display()], page 320, built on top of flimage_sdisplay(), can display single or multiple images. For typical use, [flimage_display()], page 320 or image->display should be used. [flimage_sdisplay()], page 320 is useful only if you're coding your own multi-image display routine. For example, [flimage_display()], page 320 is built roughly like the following

```
int flimage_display(FL_IMAGE *im, FL_WINDOW win) {
    int err;

    for (err = 0; err >=0 && im; im = im->next) {
        err = flimage_sdisplay(im, win);
        fl_update_display(0);
        fl_msleep(im->setup->delay);
    }

    return err;
}
```

And you can build your own multi-frame image display routine to suit your application's needs.

Despite the display routine's simple look, this function performs tasks that involve the details of dealing with different hardware capabilities, a daunting task for beginners. For PseudoColor displays (i.e., using color maps or color lookup tables), a color quantization or dithering step may be performed by the function to reduce the number of colors in the image (of course, the colorreduced image is kept only for display, the original image is untouched so future processing is carried out on the original full resolution image, rather than the displayed, an approximate of the original image). In general, when the information in an image is reduced in order to display it, the original image is not altered in any way. For example, this function can display a 24bit image on a 1bit display without losing any information on the original 24bit image.

By default, the entire image is displayed at the top-left corner of the window. To display the image at other locations within the window (perhaps to center it), use the `image->wx` and `image->wy` fields of the `FL_IMAGE` structure. These two fields specify where in the window the origin of the image should be. By repeatedly changing `image->wx` and `image->wy` and displaying, image panning can be implemented.

It is also possible to display a subimage by specifying non-zero value for `(image->sx,image->sy)` and `(image->sw, image->sh)`. You can view the image as a 2D space with the origin at the top left corner. The positive y axis of the image space is pointing downward. `(image->sx,image->sy)` specify the subimage offset into the image (they must be non-negative) and `(image->sw,image->sh)` specify the width and height of the subimage. Taken the window offset and the subimage together, the more accurate statement of the functionality of the the function `[flimage_display()], page 320` is that it displays a subimage specified by `(image->sx,image->sy)` and `(image->sw,image->sh)` starting at `(image->wx, image->wy)`.

You can also use clipping to display a subimage by utilizing the following functions and `image->gc`

```
fl_set_gc_clipping(image->gc, x, y, w, h);
fl_unset_gc_clipping(image->gc);
```

where the coordinates are window coordinates. Of course, by manipulating `image->gc` directly, more interesting clipping or masking can be achieved. Since the GC is visual dependent, a newly created image before displaying may not yet have a valid GC assoiated with it. If you must set some clipping before displaying, you can set the `image->gc` yourself beforehand. Note that you if you free the GC, make sure you reset it to `None`.

To display an image in a canvas, the following can be used

```
flimage_display(image, FL_ObjWin(canvas));
```

Since this function only knows about window IDs, and writes to the window directly, it may not be sensitive to the status of the form the canvas is on, e.g., a frozen form. In your application, you should check the status of the form before calling this function.

Sometimes it may be useful to find out if a specific file is an image file before attempting to read it (for example, as a file filter). To this end, the following routine exists

```
int flimage_is_supported(const char *file);
```

The function returns true if the specified file is a known image file. If the file is not a known image or not readable for any reason, the function return 0.

37.2 The FL_IMAGE Structure

Before we go into more details on image support, some comments on the image structure are in order. The image structure contains the following basic fields that describe fully the image in question and how it should be displayed.

```
typedef unsigned char FL_PCTYPE;         /* primary color type */
#define FL_PCBITS     8                  /* primary color bits */
#define FL_PCMAX      ((1<<FL_PCBITS)-1) /* primary color max val */
typedef unsigned int  FL_PACKED;         /* packed RGB(A) type */

typedef struct flimage_ {
    int               type;
    int               w,
                      h;
    void            * app_data;
    void            * u_vdata;
    unsigned char  ** red;
    unsigned char  ** green;
    unsigned char  ** blue;
    unsigned char  ** alpha;
    unsigned short ** ci;
    unsigned short ** gray;
    FL_PACKED      ** packed;
    short           * red_lut;
    short           * green_lut;
    short           * blue_lut;
    short           * alpha_lut;
    int               map_len;
    int               colors;
    int               gray_maxval;
    int               app_background;
    int               wx,
                      wy;
    int               sx,
                      sy;
    int               sw,
                      sh;
    char            * comments;
    int              comments_len;
    void            * io_spec;
    int               spec_size;
    int               (*display) (struct flimage_ *, FL_WINDOW win);
    struct flimage_ * next;
    int               double_buffer;
    unsigned long     pixmap;
    /* more stuff omitted */
} FL_IMAGE;
```

The meaning of each field is as follows:

`type` This field specifies the current image type and storage (1bit, 24bit etc. See next section for details). The image type also indicates implicitly which of the pixel fields should be used.

`w,h` The width and height of the image.

`app_data` A field that's initialized at image creation. Its value can be set by the application prior to any existence of image. Once set, all images created thereafter will have the same value for this field. See Section later. The Forms Library does not modify or reference it once it's initialized.

`u_vdata` A field for use by the application. This field is always initialize to null. The Forms Library does not reference or modify it.

`red, green, blue, alpha`
This first three fields are the color components of a 24 bit image, each of which is a 2-dimensional array. The 2D array is arranged so the image runs from left to right and top to bottom. For example, the 3rd pixel on the 10th row is composed of the following RGB elements: `(red[9][2],green[9][2],blue[9][2])`. Note however, these fields are meaningful only if the image type is `FL_IMAGE_RGB`. Although it's always allocated for a 24bit image, `alpha` is currently not used by the Forms Library

`ci` The field are the pixel values for a color index image (image type `FL_IMAGE_CI`). The field is also a 2-dimensional array arranged in the same way as the fields `red`, `green` and `blue`, i.e., the image runs from left to right, top to bottom. For example, `ci[3][9]` should be used to obtain the 10th pixel on the 4th row. To obtain the RGB elements of a pixel, the pixel value should be used as an index into a lookup table specified by the fields `red_lut`, `green_lut` and `blue_lut`. Although `ci` can hold an unsigned short, only the lower `FL_LUTBITS` (12) bits are supported, i.e., the color index should not be bigger than 4095.

`gray` This field, again a 2-dimensional array, holds the pixels of a gray image. The pixel values are interpreted as intensities in a linear fashion. Two types of gray images are supported, 8 bit (`FL_IMAGE_GRAY`) and 16 bit (`FL_IMAGE_GRAY16`). For 16 bit gray image, the actual depths of the image is indicated by member `gray_maxval`. For example, if `gray_maxval` is 4095, it is assumed that the actual pixel value ranges from 0 to 4095, i.e., the gray scale image is 12 bit. For 8 bit grayscale image, `gray_maxval` is not used. This means that the type `FL_IMAGE_GRAY` is always assumed to be 8 bit, the loading and creating routine should take care to properly scale data that are less than 8 bit.

`gray_maxval`
This field is meaningful only if the image type is `FL_IMAGE_GRAY16`. It specifies the actual dynamic range of the gray intensities. Its value should be set by the image loading routines if the gray image depth is more than 8 bits.

`ci_maxval`
This field by default is 256, indicating the maximum value of the color index.

`packed` This field (a 2-dimensional array) holds a 24 bit/32 bit image in a packed format. Each element of the 2D array is an unsigned integer (for now) that holds the RGB, one byte each, in the lower 24 bits of the integer. The topmost byte is not used. The macro `FL_PACK(r, g, b)` should be used to pack the triplet `(r, g, b)` into a pixel and `FL_UNPACK(p, r, g, b)` should be used to unpack a pixel. To obtain individual primary colors, the macros `FL_GETR(p)`, `FL_GETG(p)` and `FL_GETB(p)` are available.

Note that the use of the macros to pack and unpack are strongly recommended. It will isolate the application program from future changes of the primary color type (for example, 16-bit resolution for R,G and B).

`red_lut, green_lut, blue_lut, alpha_lut`
These are the lookup tables for a color index image. Each of the table is a 1D array of length `image->map len`. Although `alpha lut` is always allocated for a color index image, it's currently not used by the Forms Library.

`map_len` The length of the colormap (lookup table).

`app_background`
A packed RGB value indicating the preferred color to use for the background of an image (also known as transparent color). This field is initialized to an illegal value. Since there is no portable way to obtain the window background the application has to set this field if transparency is to be achieved. In future versions of image support, other means of doing transparency will be explored and implemented.

`wx, wy` The window offset to use to display the image.

`sx, sy, sw, sh`
The subimage to display.

`comments` This is typically set by the loading routines to convey some information about the image. The application is free to choose how to display the comment, which may have embedded newlines in it.

`io_spec` This field is meant for the reading/writing routine to place format specific state information that otherwise needs to be static or global.

`spec_size`
This field should be set to the number of bytes `io_spec` contains.

`display` A function you can use to display an image. The image loading routine sets this function.

`next` This is a link to the next image. This is how `[flimage_load()], page 319` chains multiple image together.

`double_buffer`
If true, the display function will double-buffer the image by using a pixmap. For typical image display it's not necessary to enable double-buffering as it is very expensive (memory and speed). Double-buffering may be useful in image editing.

`pixmap` The backbuffer pixmap if double-buffered.

Although it is generally not necessary for an application to access individual pixels, the need to do so may arise. In doing so, it is important to consult the `image->type` field before dereferencing any of the pixel field. That is, you should access `image->ci` only if you know that the image type is `FL_IMAGE_CI` or `FL_IMAGE_MONO`.

37.3 Supported image types

Forms Library supports all common and not-so-common image types. For example, the supported images range from the simple 1 bit bitmap to full 24 bit RGB images. 12 bit gray scale images (common in medical imaging) are also supported.

The supported image types are denoted using the following constants, all of them (except `FL_IMAGE_FLEX`) using a different bit, so they can be bitwise ORed together:

```
FL_IMAGE_MONO,    /* 1 bit bitmaps */
FL_IMAGE_GRAY,    /* gray-scale image (8 bit) */
FL_IMAGE_GRAY16,  /* gray-scale image (9 to 16 bit) */
FL_IMAGE_CI,      /* generic color index image */
FL_IMAGE_RGB,     /* 24 bit RGB(A) image */
FL_IMAGE_PACKED,  /* 24 bit RGB(A) image. Packed storage */
FL_IMAGE_FLEX,    /* All of the above */
```

For the 24 bit variety another 8 bit (`image->alpha` and the top-most byte of the packed integer) is available for the application, perhaps storing the alpha values into it. The Forms Library does not modify or reference this extra byte.

Mono (b&w) images are stored as a colormap image with a lut of length 2.

The `FL_IMAGE_FLEX` type is mainly for the reading and loading routines to indicate the types they are capable of handling. For example, if you're coding an output routine, you use `FL_IMAGE_FLEX` to indicate that the output routine can take any type the image. Otherwise the driver will convert the image type before handing the image over to the actual output routine.

In displaying an image of type `FL_IMAGE_GRAY16`, window leveling, a technique to visualize specific ranges of the data, is employed. Basically, you specify a window level (`level`) and a window width (`wwidth`) and the display function will map all pixels that fall within `level-width/2` and `level+width/2` linearly to the whole dynamic range of the intensities the hardware is capable of displaying. For example, if the display device can only display 256 shades of gray, `level-width/2` is mapped to 0 and `level+width/2` is mapped to 255, and pixels values between `level-width/2` and `level+width/2` are linearly mapped to values between 0 and 255. Pixel values that fall below `level-width/2` are mapped to zero and those that larger than `level+width/2` are mapped to 255.

Use the following routine to set the window level

```
int flimage_windowlevel(FL_IMAGE *im, int level, int wwidth);
```

The function returns 1 if window level parameters are modified, otherwise 0 is returned. Setting `wwidth` to zero disables window leveling. Note that if `im` points to a multiple image, window level parameters are changed for all images.

To obtain the image type name in string format, e.g., for reporting purposes, use the following routine

```
const char *flimage_type_name(int type);
```

To convert between different types of images, the following routine is available

```
int flimage_convert(FL_IMAGE *image, int newtype, int ncolors);
```

The parameter `newtype` should be one of the supported image types mentioned earlier in this section. Parameter `ncolors` is meaningful only if `newtype` is `FL_IMAGE_CI`. In this case, it specifies the number of colors to generate, most likely from a color quantization process. If the conversion is successful a non-negative integer is returned, otherwise a negative integaer. Depending on which quantization function is used, the number of quantized colors may not be more than 256.

To keep information loss to a minimum, `[flimage_convert()], page 326` may elect to keep the original image in memory even if the conversion is successful. For example, converting a full color image (24 bit) into a 8 bit image and then converting back can lose much information of the image if the converting function does not keep the original image.

What this means is that the following sequence gets back the original image

```
/* the current image is RGB. Now we reduce the full color
   image to 8 bit color index image. The conversion routine
   will keep the 24 bit color. */

flimage_convert(image, FL_IMAGE_CI, 256);

/* Now convert back to RGB for image processing. The con-
   version routine will notice that the input image was
   originally converted from a 24bit image. Instead of
   doing the conversion, it simply retrieves the saved
   image and returns. */

flimage_convert(image, FL_IMAGE_RGB, 0);
```

This behavior might not always be what the application wants. To override it, you can set `image->force_convert` to 1 before calling the conversion routine. Upon function return the flag is reset to zero.

37.4 Creating Images

With the basic fields in the image structure and image types explained, we're now in a position to tackle the problem of creating images on the fly. The data may have come from some simulations or some other means, the task now is to create an image from the data and try to display/visualize it.

The first task involved in creating an image is to create an image structure that is properly initialized. To this end, the following routine is available

```
FL_IMAGE *flimage_alloc(void);
```

The function returns a pointer to a piece of dynamically allocated memory that's properly initialized.

The task next is to put the existing data into the structure. This involves several steps. The first step is to figure out what type of image to create. For scalar data, there are two logical choices, either a gray-scale intensity image or a color index image with the data being interpreted as indices into some lookup table. Both of these may be useful. Gray-scale

imagse are straight forward to create and the meaning of the pixel values is well defined and understood. On the other hand with color-mapped image you can selectively enhance the data range you want to visualize by choosing appropriate color-maps. For vector data, RGB image probably makes most sense. In any case it's strictly application's decision. All that is needed to make it work with Forms Library is to set the `image->type` field to a valid value. Of course the image dimension (width and height) also needs to be set. Once this is done, we need to copy the data into the image structure.

Before we copy the data we create the destination storage using one of the following routines

```
void *fl_get_matrix(int nrows, int ncols, unsigned int elem_size);
int flimage_getmem(FL_IMAGE *image);
```

The `[fl_get_matrix()], page 327` function creates a 2-dimensional array of entities of size `elem_size`. The array is of `nrows` by `ncols` in size. The 2D array can be passed as a pointer to pointer and indexed as a real 2D arrays. The `[flimage_getmem()], page 327` routine allocates the proper amount of memory appropriate for the image type, including colormaps when needed.

After the destination storage is allocated, copying the data into it is simple

```
image->type = FL_IMAGE_GRAY;
image->w    = data_columns;
image->h    = data_row;
flimage_getmem(image);
/* or you can use the instead
  im->gray = fl_get_matrix(im->h, im->w, sizeof **im->gray);
*/

for (row = 0; row < image->h; row++)
    for (col = 0; col < image->w; col++)
        image->gray[row][col] = data_at_row_and_col;
```

Of course, if data is stored row-by-row, a `memcpy(3)` instead of a loop over columns may be more efficient. Also if your data are stored in a single array, `[fl_make_matrix()], page 347` might be a lot faster as it does not copy the data.

If the created image is a color index image, in addition to copying the data to `image->ci`, you also need to set the lookup table length `image->map_len`, which should reflect the dynamic range of the data:

```
image->type    = FL_IMAGE_CI;
image->w       = A;
image->h       = B;
image->map_len = X;
flimage_getmem(image);  /* this will allocate ci and lut */

for (row = 0; row < image->h; row++)
    for (col = 0; col < image->w; col++)
        image->ci[row][col] = data;

for (i = 0; i < image->map_len; i++) {
```

```
    image->red_lut[i]   = some_value_less_than_FL_PCMAX;
    image->green_lut[i] = some_value_less_than_FL_PCMAX;
    image->blue_lut[i]  = some_value_less_than_FL_PCMAX;
}
```

If the type is `FL_IMAGE_GRAY16`, you also need to set `image->gray_maxval` to the maximum value in the data.

Now we're ready to display the image

```
flimage_display(image, win);
```

As mentioned before, the display routine may create a buffered, display hardware specific and potentially lower-resolution image than the original image. If for any reason, you need to modify the image, either the pixels or the lookup tables, you need to inform the library to invalidate the buffered image:

```
image->modified = 1;
```

37.5 Supported Image Formats

There are many file formats for image storage. The popularity, flexibility and cleanness of the different formats varies. Forms Library supports several popular ones, but these are not the only ones that are popular. Toward the end of this section, it will be outlined how to extend the image support in the Forms Library so more image file can be read by `[flimage_load()], page 319`.

37.5.1 Built-in support

Each image file format in Forms Library is identified by any one of three pieces of information, the formal name, the short name, and the file extension. For example, for the GIF format, the formal name is `"CompuServe GIF"`[1], the short name is `"GIF"`, and file extension is `"gif"`. This information is used to specify the output format for `[flimage_dump()], page 319`.

The following table summarizes the supported file formats with comments

FormalName	ShortName	Extension	Comments
Portable Pixmap	ppm	ppm	
Portable Graymap	pgm	pgm	
Portable Bitmap	pbm	pbm	
CompuServe GIF	gif	gif	
Windows/OS2 BMP file	bmp	bmp	
JPEG/JFIF format	jpeg	jpg	
X Window Bitmap	xbm	xbm	
X Window Dump	xwd	xwd	
X PixMap	xpm	xpm	XPM3 only
NASA/NOST FITS	fits	fits	Standard FITS and IMAGE extension
Portable Network Graphics	png	png	needs netpbm

[1] The Graphics Interchange Format (c) is the Copyright property of CompuServe Incorporated. GIF(sm) is a Service Mark property of CompuServe Incorporated.

SGI RGB format	iris	rgb	need pbmplus/netpbm package
PostScript format	ps	ps	needs gs for reading
Tagged Image File Format	tiff	tif	no compression support

To avoid executable bloating with unnecessary code, only ppm, pgm, pbm and compression filters (gzip and compress) are enabled by default. To enable other formats, call `flimage_enable_xxx()` once anywhere after `[fl_initialize()], page 277`, where `xxx` is the short name for the format. For example, to enable BMP format, `flimage_enable_bmp()` should be called.

Further, if you enable GIF support, you're responsible for any copyright/patent and intellectual property dispute arising from it. Under no circumstance should the authors of the Forms Library be liable for the use or misuse of the GIF format.

Usually there are choices on how the image should be read and written. The following is a rundown of the built-in options that control some aspects of image support. Note that these options are persistent in nature and once set they remain in force until reset.

```
typedef struct {
    int quality;
    int smoothing;
} FLIMAGE_JPEG_OPTIONS;

void flimage_jpeg_output_options(FLIMAGE_JPEG_OPTIONS *option);
```

The default quality factor for JPEG output is 75. In general, the higher the quality factor rhe better the image is, but the file size gets larger. The default smoothing factor is 0.

```
void flimage_pnm_output_options(int raw_format);
```

For PNM (ppm, pgm, and pbm) output, two variants are supported, the binary (raw) and ASCII format. The raw format is the default. If the output image is of type `FL_IMAGE_GRAY16`, ASCII format is always output.

```
void flimage_gif_output_options(int interlace);
```

If `interlace` is true, an interlaced output is generated. Transparency, comments, and text are controlled, respectively, by `image->tran_rgb`, `image->comments` and `image->text`.

PostScript options affect both reading and writing.

```
FLIMAGE_PS_OPTION *flimage_ps_options(void);
```

where the control structure has the following members

`int orientation`
: The orientation of the generated image on paper. Valid options are `FLPS_AUTO`, `FLPS_PORTRAIT` and `FLPS_LANDSCAPE`. The default is `FLPS_AUTO`.

`int auto_fit`
: By default, the output image is scaled to fit the paper if necessary. Set it to false (0) to turn auto-scaling off.

`float xdpi, ydpi`
: These two are the screen resolution. Typical screens these days have resolutions about 80 dpi. The settings of these affect both reading and writing.

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

`float paper_h`
: The paper height, in inches. The default is 11.0 in

`char* tmpdir`
: A directory name where temporary working files go. The default is `/tmp`.

`float hm, vm`
: Horizontal and vertical margins, in inches, to leave when writing images. The default is 0.4 in (about 1 cm).

`float xscale`
: Default is 1.0.

`float yscale`
: Default is 1.0.

`int first_page_only`
: If set, only the first page of the document will be loaded even if the document is multi-paged. The default setting is false.

To change an option, simply call `[flimage_ps_options()], page 329` and change the field from the pointer returned by the function:

```
void SetMyPageSize(float w, float h) {
    FLIMAGE_PS_OPTION *options = flimage_ps_options();

    options->paper_w = w;
    options->paper_h = h;
}
```

All these option setting routines can be used either as a configuration routine or an image-by-image basis by always calling one of these routines before `[flimage_dump()], page 319`. For example,

```
flimage_jpeg_output_options(option_for_this_image);
flimage_dump(im, "file","jpeg");
```

You can also utilize the `image->pre_write` function to set the options. This function, if set, is always called inside `[flimage_dump()], page 319` before the actual output begins.

37.5.2 Adding New Formats

It is possible for application to add new formats to the library so `[flimage_load()], page 319` and `[flimage_dump()], page 319` know how to handle them. Basically, the application program tells the library how to identify the image format, and the image dimension, and how to read and write pixels.

The API for doing so is the following

```
typedef int (*FLIMAGE_Identify) (FILE *);
typedef int (*FLIMAGE_Description) (FL_IMAGE *);
typedef int (*FLIMAGE_Read_Pixels) (FL_IMAGE *);
typedef int (*FLIMAGE_Write_Image) (FL_IMAGE *);
```

```
int flimage_add_format(const char *formal_name,
                       const char *short_name,
                       const char *extension,
                       int type,
                       FLIMAGE_Identify identify,
                       FLIMAGE_Description description,
                       FLIMAGE_Read_Pixels read_pixels,
                       FLIMAGE_Write_Image write_image);
```

where we have

`formal_name`
: The formal name of the image format

`short_name`
: An abbreviated name for the image format

`extension`
: File extension, if this field is `NULL`, `short_name` will be substituted

`type`
: The image type. This field generally is one of the supported image types (e.g., `FL_IMAGE_RGB`), but it does not have to. For image file formats that are capable of holding more than one type of images, this field can be set to indicate this by ORing the supported types together (e.g., `FL_IMAGE_RGB|FL_IMAGE_GRAY`). However, when description returns, the image type should be set to the actual type in the file.

`identify`
: This function should return 1 if the file pointed to by the file pointer passed in is the expected image format (by checking signature etc.). It should return a negative number if the file is not recognized. The decision if the file pointer should be rewound or not is between this function and the description function.

`description`
: This function in general should set the image dimension and type fields (and colormap length for color index images) if successful, so the driver can allocate the necessary memory for read pixel. Of course, if `read_pixels` elects to allocate memory itself, the `description` function does not have to set any fields. However, if reading should continue, the function should return 1 otherwise a negative number.

  The function should read from input file stream `image->fpin`.

  It is likely that some information obtained in this function needs to be passed to the actual pixel reading routine. The easiest way is, of course, to make these information static within the file, but if a GUI system is in place, all the reading routines should try to be reentrant. The method to avoid static variables is to use the `image->io_spec` field to keep these information. If this field points to some dynamically allocated memory, you do not need to free it after `read_pixels` function finishes. However, if you free it or this field points to static memory, you should set to this field to `NULL` when finished.

  The following is a short example showing how this field may be utilized.

```
typedef struct {
```

```
    int bits_per_pixel;
    int other_stuff;
} SPEC;

static int description(FL_IMAGE *im) {
    SPEC *sp = fl_calloc(1, sizeof *sp);

    im->io_spec        = sp;
    im->spec_size      = sizeof *sp;
    sp->bits_per_pixel = read_from_file(im->fpin);

    return 0;
}

static int read_pixels(FL_IMAGE *im) {
    SPEC *sp = im->io_spec;

     int bits_per_pixel = sp->bits_per_pixel;

     read_file_based_on_bits_per_pixel(im->fpin);

     /* You don't have to free im->io_spec, but if you do
        remember to set it to NULL before returning */

     return 0;
}
```

`read_pixels`

This function reads the pixels from the file and fills one of the pixel matrix in the image structure depending on the type. If reading is successful, a non-negative number should be returned otherwise a negative number should be returned.

Upon entry, `image->completed` is set to zero.

The function should not close the file.

`write_image`

This function takes an image structure and should write the image out in a format it knows. Prior to calling this routine, the driver will have already converted the image type to the type it wants. The function should return 1 on success and a negative number otherwise. If only reading of the image format is supported this parameter can be set to `NULL`.

The function should write to file stream `image->fpout`.

By calling [`flimage_add_format()`], page 330 the newly specified image format is added to a "recognized image format" pool in the library. When [`flimage_load()`], page 319 is called the library, after verifying that the file is readable, loops over each of the formats and calls the `identify` routine until a format is identified or the pool exhausted. If the file is recognized as one of the supported formats the `description` routine is called to obtain the image dimension and type. Upon its return the library allocates all memory needed,

then calls `read_pixels`. If the image format pool is exhausted before the file is recognized `[flimage_load()], page 319` fails.

On output, when `[flimage_dump()], page 319` is called, the requested format name is used to look up the output routine from the image format pool. Once an output routine for the requested format is found, the library looks the image type the output is capable of writing. If the current image type is not among the types supported by the format the library converts image to the type needed prior to calling the output routine `write_image()`. So what `[flimage_dump()], page 319` does is

```
int flimage_dump(FL_IMAGE *im, const char *filename,
                 const char *formatName) {
    format = search_image_format_pool(formatName);
    if (!format)
        return -1;

    im->fpout = fopen(filename);
    if (im->pre_write)
        im->pre_write(im);

    convert image type if necessary(im);
    format->write_pixels(im);
    ...
}
```

If the name of the image format supplied by `[flimage_add_format()], page 330` is identical to one that is already supported, the new routines replace those that are in the pool. This way, the application can override the built-in supports.

For a non-trivial example of adding a new format, see file `flimage_jpeg.c`. Another way of adding image formats is through external filters that convert an unsupported format into one that is. All you need to do is inform the library what external filter to use. `pbmplus` or `netpbm` are excellent packages for this purpose.

The library has two functions that deal with external filters

```
int flimage_description_via_filter(FL_IMAGE * im, char *const *cmds,
                                   const char *what, int verbose);
int flimage_write_via_filter(FL_IMAGE *im, char *const *cmds,
                             char *const formats[], int verbose);
```

where `cmds` are a list of shell commands (filters) that convert the format in question into one of the supported formats. Parameter `what` is for reporting purposes and parameter `verbose` controls if some information and error messages should be printed. This is mainly for debugging purposes.

Let us go through one example to show how this filter facility can be used. In this example, we support SGI's rgb format via the `netpbm` package.

As with regular image format, we first define a function that identifies the image format:

```
static int IRIS_identify(FILE *fp) {
    char buf[2];

    fread(buf, 1, 2, fp);
```

```
    return    (buf[0] == '\001' && buf[1] == '\332')
           || (buf[0] == '\332' && buf[1] == '\001');
}
```

Then we need to define the filter(s) that can convert a RGB file into one that's supported. Here we use `sgitopnm`, but you can use diferent filters if available. Function `[flimage_description_via_filter()], page 333` will try all the filters specified until one of them succeeds. If none does an error code is returned:

```
static int IRIS_description(FL_IMAGE *im) {
    static char *cmds[] = {"sgitopnm %s > %s",
                           NULL  /* sentinel, indicating end of
                                    list of filters */ };
    return flimage_description_via_filter(im, cmds,
                                          "Reading RGB...", 0);
}
```

All commands should be suitable format strings for function `sprintf()` and contain `%s` twice. The first one will be replaced by the input file name, the second by a filename which will be supplied by the library to hold the converted image. The list must be terminate with a `NULL` element.

In the above example, `sgitopnm %s > %s` specifies the external command, `sgitopnm`, and how it operates. Basically, the library will do a `sprintf(cmdbuf, cmd[i], irisfile, tmpfile)` and then execute `cmdbuf`.

There is really no need for a load function as the filter will have already invoked the correct load function when it returns. For the record of capability queries, a dummy load function is needed:

```
static int IRIS_load(FL_IMAGE * im) {
     fprintf(stderr, "We should never get here...\n");
     return -1;
}
```

Writing an image is similar:

```
static int IRIS_dump(FL_IMAGE *im) {
    static char *cmds[] = {"pnmtosgi %s > %s",
                           NULL};
    static char *cmds_rle[] = {"pnmtosgi -rle %s > %s",
                               NULL};
    static char *formats[] = {"ppm", "pgm", "pbm", NULL};

    return flimage_write_via_filter(im, rle ? cmds_rle : cmds,
                                    formats, 0);
}
```

Again, the external commands should accept two arguments. The first argument will be supplied by the library, a temporary file that holds the converted image in a format the filter understands, and the second argument will be the requested output filename.

For output, an additional argument is required. The additional argument `formats` specifies the image format accepted by the external filter. In this case, this is the pnm format. It is

important that if the filter accepts more than one format, you should specify the formats in decreasing generality, i.e., ppm, pgm, pbm.

With these functions in place, finally we're ready to add iris support into the library

```
void add_iris(void) {
    flimage_add_format("SGI Iris", "iris", "rgb",
                       FL_IMAGE_RGB|FL_IMAGE_GRAY|FL_IMAGE_MONO,
                       IRIS_identify,
                       IRIS_description,
                       IRIS_load,
                       IRIS_dump);
}
```

After a call of `add_iris()` you can now use [`flimage_load()`], page 319 and [`flimage_dump()`], page 319 to read and write SGI iris format just like any other format.

37.5.3 Queries

Since the number of formats supported by the library is dynamic in nature, some query routines are available to obtain support information.

To obtain the number of currently supported image formats, use the routine

```
int flimage_get_number_of_formats(void);
```

The functions returns the number of formats supported, for reading or writing or both. To obtain detailed information for each format, the following can be used

```
typedef struct {
    const char * formal_name;
    const char * short_name;
    const char * extension;
    int          type;
    int          read_write;
    int          annotation;
} FLIMAGE_FORMAT_INFO;

const FLIMAGE_FORMAT_INFO *flimage_get_format_info(int n);
```

where parameter `n` is an integer between 1 and the return value of [`flimage_get_number_of_formats()`], page 335 . Upon function return a static buffer is returned containing the basic information about the image. The read_write field can be one of the following combinations thereof

`FLIMAGE_READABLE`
: supports reading

`FLIMAGE_WRITABLE`
: supports writing

or the bitwise OR of both.

These two routines are most useful for reporting or presenting capabilities to the user

```
FLIMAGE_FORMAT_INFO *info;
int n = flimage_get_number_of_formats();
```

```
fprintf(stderr,"FL supports the following format\n");
for (; n; n--) {
    info = flimage_get_format_info(n);
    fprintf(stderr,"%s format\t(%c%c)\n",
            info->short_name,
            (info->read_write & FLIMAGE_READABLE) ? 'r' : ' ',
            (info->read_write & FLIMAGE_WRITABLE) ? 'w' : ' ');
}
```

37.6 Setup and Configuration

Although the image support is designed with integration into a GUI system in mind, it neither assumes what the GUI system is nor does it need a GUI system to work. As a matter of fact, for the most part it doesn't even need an X connection to work (obviously without a connection, you won't be able to display images). For this reason, some of the typical (and necessary) tasks, such as progress and error reporting, are by default implemented only to use text output (i.e., to `stderr`). Obviously, with a GUI in place this is not quite adequate. Hooks are available for application program to re-define what to do with these tasks.

The interface to the library configuration is as follows

```
void flimage_setup(FLIMAGE_SETUP *setup);
```

where the parameter `setup` is a pointer to a structure defined as follows:

```
typedef struct {
    void       * app_data;
    int          (*visual_cue) (FL_IMAGE *im, const char *msg);
    void         (*error_message) (FL_IMAGE *im, const char *msg);
    const char * rgbfile;
    int          do_not_clear;
    int          max_frames;
    int          delay;
    int          double_buffer;
    int          add_extension;
} FLIMAGE_SETUP;
```

with

`app_data` The application can use this field to set a value so the field `image->app_data` in all image structures returned by the library will have this value. It's most useful to set this field to something that's persistent during the application run, such as the `fdui` structure of the main control panel.

Note that `image->app_data` is different from `image->u_vdata` in that all image structures returned by the library have the same value of `image->app_data`, which is set by the library. In contrast, `image->u_vdata` is set by the application on an image-by-image basis.

`visual_cue`
This is the function that will be called by all image reading, writing and processing routines. The function is meant to give the user some visual feedback

about what is happening. For lengthy tasks, this function is called repeatedly and periodically to indicate what percentage of the task is completed and to give the application program a chance to check and process GUI activities (for example, via [fl_check_forms()], page 296).

The first parameter to the function is the image currently being worked on and the second parameter is a short message, indicating the name of the task, such as "Reading JPG" etc.

Two fields in the image structure can be used to obtain progress information. The member fields `image->total` indicates the total amount of work to be done in some arbitrary units (usually number of rows in the image). `image->completed` indicates how much of the task has been completed. The percentage of how much is completed is then simply the ratio of `image->completed` and `image->total`, multiplied by 100.

At the begin of a task `image->completed` is set to a value less or equal 1, and at the end of the task, `image->completed` is set to `image->total`.

A special value of -1 for `image->completed` may be used to indicate a task of unknown length.

`error_message`

This is a function that is called when an error (of all severities) has occurred inside the library. It is recommanded that the application provide a means to show the messages to the user by sypplying this function.

The first parameter is a pointer to the image that's being worked on, and the second parameter is a brief message, such as "memory allocation failed" etc.

A convenience function, [flimage_error()], page 337, is provided to call the error message handler.

`rgbfile` This field should be set to the full path to the color name database (`rgb.txt`) if your system has it in non-standard locations. On most systems, this file is `/usr/lib/X11/rgb.txt`, which is the default if this field is not set.[2]

`do_not_clear`

By default, [flimage_display()], page 320 clears the window before displaying the image. Set this member to 1 to disable window clearing.

`no_auto_extension`

By default, [flimage_dump()], page 319 changes the filename extension to reflect the format. Set this member to 1 to disable extension substitution.

`double_buffer`

If set, all image display will by default double-buffered. Double-buffering an image is very expensive (in terms of both resource and speed) as the backbuffer is simulated using a pixmap. If there are no annotations, double-buffering an image does not really improve anything.

It is far better to turn double-buffering on and off on a image-by-image basis using the `image->double_bufffer` field.

[2] The routine where this field is used searches some more locations than the default and should work on most systems automagically.

`max_frames`
: This field specifies the maximum number of frames to read by [flimage_load()], page 319. The default maximum is 30 frames.

`delay`
: This field specifies the delay (in milliseconds) between successive frames. It is used by the [flimage_display()], page 320 routine.

Note that it is always a good idea to clear the setup structure before initializing and using it

```
FLIMAGE_SETUP mysetup;
memset(mysetup, 0, sizeof mysetup);

mysetup.max_frames = 100;
mysetup.delay      = 10;

flimage_setup(&mysetup);
```

It is possible to modify the image loading process by utilizing the following routines [flimage_load()], page 319 is based on:

```
FL_IMAGE *flimage_open(const char *name);
```

This function takes a file name and returns an image sturcture pointer if the file is a recognized image file. Otherwise `NULL` is returned.

The function

```
FL_IMAGE *flimage_read(FL_IMAGE *im);
```

takes an image structure returned by [flimage_open()], page 338 and fills the image structure. Between [flimage_open()], page 338 and [flimage_read()], page 338 you can inspect or modify fields in the image structure.

```
int flimage_close(FL_IMAGE *im);
```

This function closes all file streams used to create the image.

37.7 Simple Image Processing

Some simple image processing capabilities are present in the Forms Library image support. All the image processing routines take an image as a parameter and process it in place. If appropriate, only the subimage specified by `(image->subx, image->suby)` and `(image->subw, image->subw)` is affected (note these are different fields from those for subimage displaying). The subimage fields are best set via user interaction, perhaps by having a rubber band that the user can drag to set the size.

In the following, each routine will be briefly explained.

37.7.1 Convolution

Convolution or filtering can be done easily using the following routine

```
int flimage_convolve(FL_IMAGE *im, int **kernel,
                     int krow, int kcol);
```

This function takes a convolution kernel of `krow` by `kcol` and convolves it with the image. The result replaces the input image. The kernel size should be odd. If successful, the

function returns a positive integer, otherwise a negative number. The kernel should be allocated by [fl_get_matrix()], page 327. To use a kernel that's a C 2-dimensional array (cast to a pointer to int), use the following function

```
int flimage_convolvea(FL_IMAGE *im, int *kernel,
                      int krow, int kcol);
```

The difference between these two functions is in their usage syntax:

```
int **kernel1 = fl_get_matrix(sizeof **kernel, n, m);
int kernel2[n][m];
kernel1[x][y] = z;
kernel2[x][y] = z;
flimage_convolve(im, kernel1, n, m);
flimage_convolvea(im, (int*) kernel2, n, m); /* note the cast */
```

Two special built-in kernels are designated with the following symbolic constants

`FLIMAGE_SMOOTH`
: indicates a 3 by 3 smoothing kernel

`FLIMAGE_SHARPEN`
: indicates a 3 by 3 sharpening kernel

37.7.2 Tint

Tint as implemented in the Forms Library emulates the effect of looking at an image through a piece of colored glass. You can specify the color and transparency of the glass:

```
int flimage_tint(FL_IMAGE *im, unsigned int packed, double opacity);
```

where the parameter `packed` is a packed RGB color, specifying the color of the glass. `opacity` specifies how much the color of the image is absorbed by the glass. A value of 0 means the glass is totally transparent, i.e., the glass has no effect3, while a value of 1.0 means total opaqueness, i.e., all you see is the color of the glass. Any value between these two extremes results in a color that is a combination of the pixel color and the glass color. For example, to tint a part of the image bluish, you can set `packed` to `FL_PACK(0,0,200)` and use an opacity of 0[3].

Tint is most useful in cases where you want to put some annotations on the image, but do not want to use a uniform and opaque background that completely obscures the image behind. By using tint, you can have a background that provides some contrast to the text, yet not obscures the image beneath completely.

Tint operation uses the subimage settings.

37.7.3 Rotation

Image rotation can be easily done with the following routine

```
int flimage_rotate(FL_IMAGE *im, int angle, int subpixel);
```

where `angle` is the angle in one-tenth of a degree (i.e., a 45 degree rotation should be specified as 450) with a positive sign for counter-clock rotation. The parameter `subpixel` should be one of the following, specifying if subpixel sampling should be enabled. It can be set to either `FLIMAGE_NOSUBPIXEL` or `FLIMAGE_SUBPIXEL`.

[3] Strictly speaking, a piece of glass that is totally transparent can't have colors.

If subpixel sampling is enabled, the resulting image pixels are interpolated from the original pixels. This usually has an "anti-aliasing" effect that leads to less severe jagged edges and similar artifacts commonly encountered in rotations. However, it also means that a color indexed image gets converted to a RGB image. If preserving the pixel value is important, you should not turn subpixel sampling on.

`[flimage_rotate()], page 339` return a negative number if it for some reason (usually due to running out of memory) fails to perform the rotation.

Since the rotated image has to be on a rectangular grid, the regions that are not occupied by the image are filled with a fill color, where the default is black. If a different fill color is desired you can set the `image->fill_ccolor` field to a packed RGB color before calling the rotation function. Note that even for color indexed images the fill color should be specified in RGB. The rotation function will search the colormap for the appropriate index if no subpixel sampling is used.

Repeated rotations should be avoided if possible. If you have to call it more than once it's a good idea to crop after rotations in order to get rid of the regions that contain only fill color.

37.7.4 Image Flipping

Image flipping refers to the mirror operation in x- or y-direction at the center. For example, to flip the columns of an image, the left and right of the image are flipped (just like having a vertical mirror in the center of the image) thus the first pixel on any given row becomes the last, and the last pixel becomes the first etc.

The API for flipping is as follows

```
int flimage_flip(FL_IMAGE *im, int what);
```

where `what` can be `'c'` or `'r'`. indicating if column and row flipping is desired.

37.7.5 Cropping

There are two functions available to crop an image

```
int flimage_autocrop(FL_IMAGE *im, unsigned int background);
int flimage_crop(FL_IMAGE *im, int xl, int yt, int xr, int yb);
```

The first function, as its name suggests, automatically crops an image using the background as the color to crop. The function works by searching the image from all four sides and removing all contiguous regions of the uniform background from the sides. The image is modified in place. If cropping is successful, a non-negative integer is returned, otherwise -1. If `background` is specified as the constant `FLIMAGE_AUTOCOLOR`, the background is chosen as the first pixel of the image.

The second function uses the parameters supplied by the user to crop the image. `xl` and `xr` are the offsets into the image from the left and the right sides, respectively, i.e., if both `xl` and `xr` are 1, the cropping removes the first column and the last column from the image. Parameters `yt` and `yb` specify the offsets into the image from the top and bottom of the image respectively.

Note the offsets do not have to be positive. When they are negative, they indicate enlargement of the image. The additional regions are filled with the uniform color specified by `image->fill_color`, a packed RGB color. This can be quite useful to add a couple of

pixels of border to an image. For example, the following adds a 1 pixel wide yellow border to an image

```
image->fill_color = FL_PACK(255,255,0);
flimage_crop(image, -1, -1, -1, -1);
```

Another function is available that can be used to obtain the auto-cropping offsets

```
int flimage_get_autocrop(FL_IMAGE *im, unsigned background,
                         int *xl, int *yt, int *xl, int *yb);
```

This function works the same way as `[flimage_autocrop()], page 340`, except that no actual cropping is performed. Upon function return the parameters `xl`, `yt`, `xl` and `yb` are set to the offsets found by the function. The application can then make adjustment to these offsets and call `[flimage_crop()], page 340`.

37.7.6 Scaling

An image can be scaled to any desired size with or without subpixel sampling. Without subpixel sampling simple pixel replication is used, otherwise a box average algorithm is employed that yields an anti-aliased image with much less artifacts. A special option is available that scales the image to the desired size but keeps the aspect ratio of the image the same by filling the part of the image that would otherwise be empty.

The main entry point to the scaling function is

```
int flimage_scale(FL_IMAGE *im, int newwidth, int newheight,
                  int option);
```

where the parameters `newwidth` and `newheight` specify the desired image size. Parameter `option`q can be one of the following constants or the bitwise OR of them:

`FLIMAGE_NOSUBPIXEL`
: scale the image with no subpixel sampling

`FLIMAGE_SUBPIXEL`
: scale the image with subpixel sampling

`FLIMAGE_ASPECT`
: scale the image with no aspect ratio change

`FLIMAGE_CENTER`
: center the scaled image if aspect

`FLIMAGE_NOCENTER`
: do not center the scaled image

For example, `FLIMAGE_ASPECT|FLIMAGE_SUBPIXEL` requests fitting the image to the new size with subpixel sampling. `FLIMAGE_ASPECT` specifies a scaling that results in an image of the requested size (even if the scales are different for width and height) without changing the aspect ratio of the original image by filling in the stretched regions with the fill color `image->fill_color`, a packed RGB color:

```
im->fill_color = FL_PACK(255,0,0);
flimage_scale(im, im->w+2, im->h, FLIMAGE_SUBPIXEL|FLIMAGE_ASPECT);
```

This code generates an image that is two pixels wider than the original image but with the same aspect ratio. The two additional pixel columns on each side of the image are filled

with the fill color (red), yielding a red border. The fitting can be useful in turning a series of images of unequal sizes into images of equal sizes with no perceptible change in image quality.

Depending on what the application requires, simple scaling (zooming) with no subpixel sampling is much faster than box averaging or blending, but subpixel sampling tends to yield smoother images with less scaling artifacts.

37.7.7 Warping

Image warping (or texture mapping in 2D) refers to the transformation of pixel coordinates. Rotation, scaling and shearing etc. are examples of (linear and non-perspective) image warping. In typical applications some of the commonly used pixel coordinate transformations are implemented using more efficient algorithms instead of a general warping. For example, image rotation is often implemented using three shears rather than a general warp (Forms Library implements rotation via image warping).

Non-perspective linear image warping in general is characterized by a 2x2 warp matrix `W` and a translation vector `T` with two elements as follows

```
P' = W * P + T
```

where `P` is a vector describing a position via it's x and y coordinates and `P'` is the position after warping.

The elements `w[i][j]` of the warp matrix are constants (if the warp matrix isn't constant or is of higher order, we usually call such a transformation morphing rather than warping). Since our destination for the warped image is an array of pixels rather than a properly defined coordinate system (such as a window) the translation has no meaning. For the following discussion, we assume the translation vector is zero. (In doing the actual warping, the warped image is indeed shifted so it starts at the (0,0) element of the array representing it).

Although, theoretically, any 2D matrix can be used as a warp matrix, there are practical constraints in image warping due to the discreteness of pixel coordinates. First of all, we have to snap all pixel coordinates onto a 2D rectangular integer grid. This in general will leave holes in the warped image because two pixels may get mapped to a single destination location, leaving a hole in the destination image. Secondly, truncation or rounding the resulting floating point values introduces errors. Because of these reasons, image warping is performed in reverse. That is, instead of looping over all pixel coordinates in the original image and transforming those into new coordinates, we start from the new coordinates and use inverse warping to obtain the coordinates of the pixel in the original image. This requires that the inverse of the warp matrix must exist (which is the case if `w[0][0] * w[1][1] != w[0][1] * w[1][0]`, i.e., the warp matrix has a non-vanishing determinante). With inverse warping the transformation becomes a re-sampling of the original image, and subpixel sampling (anti-aliasing) can be easily implemented.

The following function is available in the library to perform warping

```
int flimage_warp(FL_IMAGE *im, float matrix[][2],
                 int neww, int newh, int subpixel);
```

where `matrix` is the warp matrix. `neww` and `newh` specify the warped image size. To have the warp function figure out the minimum enclosing rectangle of the warped image you can pass zeros for the new width and height. Nevertheless, you can specify whatever size you

want and the warp function will fill the empty grid location with the fill color. This is how the aspect ratio preserving scaling is implemented.

In general, the warped image will not be rectangular in shape. To make the image rectangular the function fills the empty regions. The fill color is specified by setting the `image->fill_color` field with a packed RGB color.

The last argument, `subpixel` specifies if subpixel sampling should be used. Although subpixel sampling adds processing time, it generally improves image quality significantly. The valid values for this parameter is any logical OR of `FLIMAGE_NOSUBPIXEL`, `FLIMAGE_SUBPIXEL` and `FLIMAGE_NOCENTER`.

`FLIMAGE_NOCENTER` is only useful if you specify an image dimension that is larger than the warped image, and in that case the warped image is flushed top-left within the image grid, otherwise it is centered.

To illustrate how image warping can be used, we show how an image rotation by an angle `deg` can be implemented:

```
float m[2][2];
m[0][0] = m[1][1] = cos(deg * M_PI / 180.0);
m[0][1] = sin(deg * M_PI / 180.0);
m[1][0] = -m[0][1];

flimage_warp(im, mat, 0, 0, FLIMAGE_SUBPIXEL);
```

Please note that the transformation is done in-place, i.e., after the function returns the image structure pointer, `im`, points to the rotated image.

If you specify a warp matrix with the off-diagonal elements being zero (scaling matrix), the image will only be scaled (in x-direction by `m[0][0]` and in y-direction by `m[1][1]`) without being also rotated.

By experimenting with various warp matrices you can obtain some interesting images. Just keep in mind that large values of the warp matrix elements tend to make the final image larger.

37.7.8 General Pixel Transformation

Many image processing tasks can be implemented as seperate RGB transformations. These transformations can be done very efficiently through the use of lookup tables. For this reason the following routine exists:

```
int flimage_transform_pixels(FL_IMAGE *im, int *red,
                             int *green, int *blue);
```

where `red`, `green` and `blue` are the lookup tables of a length of at least `FL_PCMAX + 1` (typically 256). The function returns a postive number on success and the image will be replaced. Note that this routine notices the settings of the subimage, i.e., you can transform a portion of the image.

To illustrate the use of this routine let's look at how a simple contrast adjustment may be implemented:

```
#include <forms.h>
#include <math.h>
```

```
int AdjustContrast(FL_IMAGE *im) {
    int r[FL_PCMAX+1],
        g[FL_PCMAX+1],
        b[FL_PCMAX+1];
    int i,
        scale = 10;

    /* in this example rgb are adjusted the same way */
    for ( i = 0; i <= FL_PCMAX; i++)
        r[i] = g[i] = b[i] = i * log10(1 + i * scale / FL_PCMAX )
                             / log10( 1 + scale );

    return flimage_transform_pixels(im, r, g, b);
}
```

37.7.9 Image Annotation

You can annotate an image with text or simple markers (arrows etc.). The location of the annotation can either be in pixel coordinate system or some application defined coordinate system.

37.7.9.1 Using Text Strings

To place text into the image, use the following routine

```
int flimage_add_text(FL_IMAGE *im, const char *str, int len,
                     int fstyle, int fsize, unsigned tcolor,
                     unsigned bcolor, int nobk, double tx,
                     double ty, int rotation);
```

where `fstyle` and `fsize` are the same as the label font style and size defined earlier in Section 3.11.3. `tcolor` and `bcolor` specify the colors to use for the text `str` and the background if the `nobk` argument is false. If `nobk` is true the text is drawn without a background. `tx` and `ty` specify the location of the text relative to the image origin. The location specified is the lower-right corner of the text. Note that the location specified can be in some physical space other than pixel space. For example, if the pixel-pixel distance represents 10 miles on a map, you'd like to be able to specify the text location in miles rather than pixels. The location is converted into pixel space using the following code

```
tx_pixel = im->xdist_scale * tx + im->xdist_offset;
ty_pixel = im->ydist_scale * ty + im->ydist_offset;
```

By default, the offsets `im->xdist_offset` and `im->yxdist_offset` are initialized to 0 and the scales `im->xdist_scale` and `im->ydist_scale` to 1.

The function returns the current number of strings for the image. The interpretation of text used also used elsewhere applies, i.e., if `str` starts with character `@` a symbol is drawn.

There is another function, maybe more convenient depending on the application, that you can use

```
int flimage_add_text_struct(FL_IMAGE *im,
                            const FLIMAGE_TEXT *text);
```

With this function instead of passing all the parameters individual;y you pass a `FLIMAGE_TEXT` structure to the function. The structure has the following fields:

`str` The string to append to the image.

`len` Length of the string in bytes.

`x, y` A location relative to the image origin, given in pixels (no conversion from other coordinate systems is done)

`align` Specifies the alignment of the string relative to the give location.

`style, size`
The font style and size to use.

`color` The text color

`bcolor` The background color

`nobk` If true indicates that no background is to be drawn.

`angle` Angle (in thenth of a degree) the text is to be rotated from the default horizontal orientation. Currently only PostScript output handles this correctly.

To delete the all texts you added to an image, use

```
void flimage_delete_all_text(FL_IMAGE *im);
```

You also can suppress the display of annotation text without deleting it. To do this, simply set `im->dont_display_text` to true.

37.7.9.2 Using Markers

In addition to text strings you can also add simple markers (arrows, circles etc) to your image.

To add a marker to an image use the following routines

```
int flimage_add_marker(FL_IMAGE *im, const char *name,
                       double x, double y, double w, double h,
                       int linestyle, int fill, int rotation,
                       FL_COLOR, FL_COLOR bcol);
int flimage_add_marker_struct(FL_IMAGE *im, const FLIMAGE_MARKER *m);
```

where `name` is the marker name (see below for a list of built-in markers). The marker name must consist of regular ASCII characters. `linestyle` indicates the line style (`FL_SOLID`, `FL DOT` etc., see Chapter 27 for a complete list. `fill` indicates if the marker should be filled or not. `x` and `y` are the coordinates of the center of the marker in physical coordinates (i.e., the same transformation as described above for annotated texts is applied), `w` and `h` are the size of the bounding box of the marker, again in physical coordinates. Every marker has a natural orientation from which you can rotate it. The angle of rotation is given by `rotation` in tenth of a degree. `col` is the color of the marker, in packed RGB format. `bcol` is currently un-used.

The second function takes a structure that specifies the marker. The members of the structure are as follows:

`name` The name of the marker.

`x, y` Position of center of the marker in pixel coordinates, relative to the origin of the image.

`w, h` The size of the bounding box in pixel coordinates.

`color` The color of the marker in packed RGB format.

`fill` If true the marker is filled.

`thickness`
The line line thickness used for drawing.

`style` The line style to be used for drawing.

`angle` Angle of rotation in tenth of a degree from the marker's nature orientation.

If successful both functions return the number of markers that are currently associated with the image, otherwise a negative number.

Some built-in markers in different orientations are shown in Fig. 22.1.

To delete all markers added to an image use the function

```
void flimage_delete_all_markers(FL_IMAGE *im);
```

Of course the library would not be complete without the ability for applications to define new markers. The following function is provided so you can define your own markers:

```
int flimage_define_marker(const char *name,
                          void (*draw) (FLIMAGE_MARKER *marker),
                          const char *psdraw);
```

When the marker is to be drawn the function `draw()` is called with the marker structure. In addition to the fields listed above the following fields are filled by the library to facilitate the operation of drawing the marker

`display` The display to be drawn on.

`gc` The GC to be used in drawing

`win` The window to draw to.

`psdraw` A string that draws a marker in a square with the corner coordinates (-1, -1), (-1, 1), (1, 1) and (1, -1) in PostScript. For example the rectangle marker has the following `psdraw` string:

```
-1 -1 moveto
-1  1 lineto
 1  1 lineto
 1 -1 lineto
 closepath
```

Defining new markers is the preferred method of placing arbitary drawings onto an image as it works well with double-buffering and pixelization of the markers.

37.7.9.3 Pixelizing the Annotation

Annotations placed on the image are kept seperate from the image pixels themselves. The reasons for doing so are twofold. First, keeping the annotation seperate makes it possible to later edit the annotations. The second reason is that typically the screen has a lower resolutions than other output devices. By keeping the annotations seperate from the pixels makes it possible to obtain better image qualities when the annotations are rendered on higher-resolution devices (for example a PostScript printer).

If for some reason making the annotations a part of the image pixels is desired, use the following routine

```
int flimage_render_annotation(FL_IMAGE *image, FL_WINDOW win);
```

The function returns -1 if an error occurs. The parameter `win` is used to create the appropriate pixmap. After the function returns the annotations are rendered into the image pixels (thus an annotation or a part of it that was outside of the image is lost). Note that during rendering the image type may change depending on the capabilities of `win`. Annotations that were kept separately are deleted. Note that the image must have been displayed at least once prior to calling this function for it to work correctly.

You can always enlarge the image first via the cropping function with some solid borders. Then you can put annotation outside of the original image but within the enlarged image.

Not all image formats support the storage of text and markers. This means if you attempt to save an image that has associated text and markers into an image format that does not support it, you may lose the annotation. All pnm formats supports the storage of annotations. To find out if a particular format supports annotation storage, look at the annotation field of the `FLIMAGE_FORMAT_INFO` structure. A zero value indicates it does not support it.

37.7.10 Write Your Own Routines

The only communication required between an image processing routine and the rest of the image routines is to let the display routine know that the image has been modified by setting `image->modified` to 1. This information is used by the display routine to invalidate any buffered displayable images that were created from the original image. After displaying, `image->modified` is reset by the display routine.

37.8 Utilities

In the following some of the utilities that may come in handy when you're writing image manipulation routines are described.

37.8.1 Memory Allocation

To create a matrix to be used in several of the functions listed above use either [`fl_get_matrix()`], page 327 described above or

```
void *fl_make_matrix(int nrow, int ncol, unsigned int esize,
                     void *inMem);
```

where `nrow` and `ncol` are the number of rows and columns of the matrix respectively. `esize` is the size (in bytes) of each matrix element.

Both functions return a two-dimensional array of entities of size `esize`. The first function initializes all elements to zero. The second function does not allocate nor initialize memory for the matrix itself. Instead it uses the memory with address `inMem` that is supplied by the caller, which should be a one-dimensional array of length `nrow * ncol * esize`.

You can use the returned pointer as a regular two-dimensional array (`matrix[r][c]`) or as a single array of length `nrow *ncol`, starting from at `matrix[0]`:

```
short **matrix = fl_get_matrix(nrow, ncol, sizeof **matrix);

/* access the matrix as a 2-d array */
matrix[3][4] = 5;

/* or access it as 1D array */
*(matrix[0] + 3 * ncol + 4) = 5;

/* most useful in image processing to use it as 1D array */

memcpy(saved, matrix, nrow * ncol * sizeof **matrix);
```

To free a matrix allocated using one the above functions, use

```
void fl_free_matrix(void *matrix);
```

The function frees all memory allocated. After the function returns the matrix cab not be de-referenced anymore. In the case where the matrix was created by `[fl_make_matrix()], page 347` the function will only free the memory that's allocated to hold the matrix indices but not the memory supplied by the caller. It is the caller's responsibility to free that part of the memory.

There are also some useful functions that manipulate images directly. The following is a brief summary of them.

```
FL_IMAGE *flimage_dup(FL_IMAGE *im);
```

This function duplicates an image `im` and returns the duplicated image. At the moment, only the first image is duplicated even if the input image has multiple frames. Furthermore, markers and annotations are not duplicated.

```
Pixmap flimage_to_pixmap(FL_IMAGE *im, FL_WINDOW win);
int flimage_from_pixmap(FL_IMAGE *im, Pixmap pixmap);
```

The first function converts an image into a Pixmap (a server side resource) that can be used in the pixmap object (see pixmap-class???).

The second function does the reverse. `im` must be a properly allocated image.

37.8.2 Color Quantization

In order to display a RGB image on a color-mapped device of limited depth, the number of colors in the original image will have to be reduced. Color quantization is one way of doing this.

Two color quantization algorithms are available in the Forms Library. One uses Heckbert's median cut algorithm followed by Floyd-Steinberg dithering after which the pixels are mapped to the colors selected. The code implementing this is from the Independent

JPEG Group's two pass quantizer (`jquant2.c` in the IJG's distribution), which under copyright (c) 1991-1996 by Thomas G. Lane and the IJG.

Another method is based on the Octree quantization algorithm with no dithering and is implemented by Steve Lamont (`spl@ucsd.edu`) and is under vopyright (c) 1998 by Steve Lamont and the National Center for Microscopy and Imaging Research. This quantization library is available from `ftp://ncmir.ucsd.edu/pub/quantize/libquantize.html`. The quantizer based on this library is not compiled into the image support. The source code for using this quantizer is in `image` subdirectory.

By default, the median cut algorithm is used. You can switch to the octree based algorithm using the following call

```
void flimage_select_octree_quantizer(void);
```

To switch back to the median cut quantizer use

```
void flimage_select_mediancut_quantizer(void);
```

The median-cut quantizer tends to give better images because of the dithering step. However, in this particular implementation, the number of quantized colors is limited to 256. There is no such limit with the octree quantizer implementation.

37.8.3 Remarks

See `itest.c` and `ibrowser.c` for example use of the image support in Forms Library. `iconvert.c` is a program that converts between different file formats and does not require an X connection.

Due to access limitations, not all combinations of display depth and bits per pixel (bpp) are tested. Depths of 1 bit (1 bpp), 4 bits (8 bpp), 8 bits (8 bpp), 16 bits (16 bpp), 24 bits (32 bpp), 30 bits (32 bpp) were tested. Although it works in 12 bit PseudoColor mode, due to limitations of the default quantizer the display function does not take full advantage of the larger lookup table. Special provisions were made so a gray12 image will be displayed in 4096 shades of gray if the hardware supports 12-bit grayscale.

If JPEG support (`image_jpeg.c`) is not compiled into the Forms Library, you can obtain the jpeg library source from `ftp://ftp.uu.net/graphics/jpeg`.

Index of Functions

fl_activate_all_forms() 48, 296
fl_activate_event_callbacks() 53, 297
fl_activate_form() 48, 296
fl_activate_glcanvas() 201
fl_activate_object() 23, 297
fl_add_bitmap() 113
fl_add_bitmapbutton() 120
fl_add_box() 15, 109
fl_add_browser() 169
fl_add_browser_line() 171
fl_add_browser_line_f() 171
fl_add_button() 18, 120
fl_add_button_class() 269
fl_add_canvas() 197
fl_add_canvas_handler() 197
fl_add_chart() 117
fl_add_chart_value() 118
fl_add_checkbutton() 120
fl_add_child() 253
fl_add_choice() 219
fl_add_clock() 116
fl_add_counter() 139
fl_add_dial() 134
fl_add_event_callback() 52, 297
fl_add_formbrowser() 182
fl_add_frame() 110
fl_add_free() 56
fl_add_fselector_appbutton() 77
fl_add_glcanvas() 200
fl_add_input() 20, 147
fl_add_io_callback() 54
fl_add_labelbutton() 120
fl_add_labelframe() 111
fl_add_lightbutton() 120
fl_add_menu() 222
fl_add_nmenu() 163
fl_add_nmenu_items() 164
fl_add_nmenu_items2() 167
fl_add_object() 35, 286
fl_add_pixmap() 114
fl_add_pixmapbutton() 120
fl_add_positioner() 136
fl_add_round3dbutton() 120
fl_add_roundbutton() 120
fl_add_scrollbar() 131
fl_add_scrollbutton() 120
fl_add_select() 157
fl_add_select_items() 157
fl_add_signal_callback() 298
fl_add_slider() 19, 127
fl_add_spinner() 142
fl_add_symbol() 34
fl_add_tabfolder() 178
fl_add_text() 17, 112
fl_add_thumbwheel() 144
fl_add_timeout() 46, 300
fl_add_timer() 186
fl_add_valslider() 20, 127
fl_add_xyplot() 188
fl_add_xyplot_overlay() 193
fl_add_xyplot_overlay_file() 193
fl_add_xyplot_text() 194
fl_addto_browser() 171
fl_addto_browser_chars() 171
fl_addto_browser_chars_f() 171
fl_addto_browser_f() 171
fl_addto_choice() 220
fl_addto_choice_f() 220
fl_addto_command_log() 74
fl_addto_command_log_f() 74
fl_addto_form() 35, 286
fl_addto_formbrowser() 183
fl_addto_group() 22, 286
fl_addto_menu() 224
fl_addto_selected_xevent() 53
fl_addto_tabfolder() 179
fl_addtopup() 229
fl_adjust_form_size() 284
fl_app_signal_direct() 298
fl_arc() 260
fl_arcf() 260
fl_bgn_form() 15, 285
fl_bgn_group() 21, 285
fl_bk_color() 255
fl_call_object_callback() 52, 290
fl_calloc() 241
fl_canvas_yield_to_shortcut() 200
fl_check_command() 73
fl_check_forms() 47, 296
fl_check_only_forms() 296
fl_circ() 260
fl_circbound() 260
fl_circf() 260
fl_clear_browser() 171
fl_clear_canvas() 200
fl_clear_chart() 118
fl_clear_choice() 220
fl_clear_command_log() 74
fl_clear_menu() 225
fl_clear_nmenu() 168
fl_clear_select() 162
fl_clear_xyplot() 195
fl_color() 255
fl_create_animated_cursor() 306
fl_create_bitmap_cursor() 306
fl_create_colormap() 199
fl_create_from_bitmapdata() 113
fl_create_from_pixmapdata() 116
fl_create_generic_button() 268
fl_current_event() 290

fl_current_pup() ... 231
fl_dashedlinestyle() ... 262
fl_deactivate_all_forms() ... 48, 296
fl_deactivate_form() ... 48, 296
fl_deactivate_object() ... 22, 297
fl_default_window() ... 254
fl_defpup() ... 227
fl_delete_browser_line() ... 172
fl_delete_choice() ... 220
fl_delete_folder() ... 181
fl_delete_folder_byname() ... 181
fl_delete_folder_byname_f() ... 181
fl_delete_folder_bynumber() ... 181
fl_delete_formbrowser() ... 183
fl_delete_formbrowser_bynumber() ... 183
fl_delete_menu_item() ... 224
fl_delete_nmenu_item() ... 167
fl_delete_object() ... 35, 286
fl_delete_select_item() ... 161
fl_delete_symbol() ... 35
fl_delete_xyplot_overlay() ... 194
fl_delete_xyplot_text() ... 194
fl_deselect_browser() ... 173
fl_deselect_browser_line() ... 173
fl_diagline() ... 261
fl_disable_fselector_cache() ... 76
fl_do_forms() ... 43, 296
fl_do_only_forms() ... 296
fl_dopup() ... 230
fl_draw_object_label() ... 264
fl_draw_object_label_outside() ... 264
fl_draw_symbol() ... 34
fl_drawmode() ... 262
fl_drw_box() ... 263
fl_drw_frame() ... 263
fl_drw_text() ... 263
fl_drw_text_beside() ... 263
fl_drw_text_cursor() ... 264
fl_end_all_command() ... 73
fl_end_command() ... 73
fl_end_form() ... 15
fl_end_form(); ... 285
fl_end_group() ... 21, 285
fl_enumerate_fonts() ... 29
fl_exe_command() ... 73
fl_find_formbrowser_form_number() ... 184
fl_finish() ... 285
fl_fit_object_label() ... 284
fl_flip_yorigin() ... 282
fl_for_all_objects() ... 42
fl_form_is_activated() ... 296
fl_form_is_iconified() ... 295
fl_form_is_visible() ... 42, 296
fl_free() ... 241
fl_free_colors() ... 26, 256
fl_free_dirlist() ... 79
fl_free_form() ... 36, 286
fl_free_matrix() ... 348
fl_free_object() ... 35, 286
fl_free_pixels() ... 256
fl_free_pixmap() ... 116
fl_free_pixmap_focus_pixmap() ... 126
fl_free_pixmap_pixmap() ... 115
fl_free_pixmapbutton_pixmap() ... 125
fl_freepup() ... 231
fl_freeze_all_forms() ... 289
fl_freeze_form() ... 32, 289
fl_get_active_folder() ... 179
fl_get_active_folder_name() ... 179
fl_get_active_folder_number() ... 179
fl_get_align_xy() ... 264
fl_get_app_mainform() ... 294
fl_get_app_resources() ... 309
fl_get_border_width() ... 281
fl_get_browser() ... 173
fl_get_browser_dimension() ... 177
fl_get_browser_line() ... 172
fl_get_browser_line_yoffset() ... 174
fl_get_browser_maxline() ... 173
fl_get_browser_rel_xoffset() ... 174
fl_get_browser_rel_yoffset() ... 174
fl_get_browser_screenlines() ... 173
fl_get_browser_scrollbar_repeat() ... 177
fl_get_browser_topline() ... 173
fl_get_browser_xoffset() ... 174
fl_get_browser_yoffset() ... 174
fl_get_button() ... 19, 124
fl_get_button_mouse_buttons() ... 124
fl_get_button_numb() ... 124
fl_get_canvas_colormap() ... 199
fl_get_canvas_depth() ... 199
fl_get_canvas_id() ... 198
fl_get_char_height() ... 258
fl_get_char_width() ... 258
fl_get_chart_bounds() ... 119
fl_get_choice() ... 220
fl_get_choice_item_mode() ... 221
fl_get_choice_item_text() ... 221
fl_get_choice_maxitems() ... 221
fl_get_choice_text() ... 220
fl_get_clipping() ... 257
fl_get_clock() ... 117
fl_get_cmdline_args() ... 279
fl_get_command_log_fdstruct() ... 74
fl_get_coordunit() ... 280
fl_get_counter_bounds() ... 140
fl_get_counter_min_repeat() ... 141
fl_get_counter_precision() ... 140
fl_get_counter_repeat() ... 141
fl_get_counter_speedjump() ... 141
fl_get_counter_step() ... 140
fl_get_counter_value() ... 140
fl_get_decoration_sizes() ... 295
fl_get_dial_angles() ... 135
fl_get_dial_bounds() ... 135
fl_get_dial_direction() ... 135

fl_get_dial_step() 135
fl_get_dial_value() 135
fl_get_directory() 77
fl_get_dirlist() 79
fl_get_display() 255
fl_get_drawmode(() 263
fl_get_filename() 77
fl_get_focus_object() 21, 290
fl_get_folder() 179
fl_get_folder_area() 181
fl_get_folder_name() 179
fl_get_folder_number() 179
fl_get_fontstruct() 259
fl_get_form_background_color() 285
fl_get_form_mouse() 256
fl_get_form_vclass() 254
fl_get_formbrowser_area() 185
fl_get_formbrowser_form() 184
fl_get_formbrowser_numforms() 183
fl_get_formbrowser_topform() 185
fl_get_formbrowser_xoffset() 185
fl_get_formbrowser_yoffset() 185
fl_get_fselector_fdstruct() 78
fl_get_fselector_form() 78
fl_get_glcanvas_attributes() 201
fl_get_glcanvas_context() 201
fl_get_glcanvas_defaults() 200
fl_get_glcanvas_xvisualinfo() 201
fl_get_global_clipping() 257
fl_get_icm_color() 25, 284
fl_get_input() 21, 152
fl_get_input_color() 154
fl_get_input_cursorpos() 153
fl_get_input_editkeymap() 156
fl_get_input_format() 151
fl_get_input_numberoflines() 154
fl_get_input_screenlines() 154
fl_get_input_scrollbarsize() 154
fl_get_input_selected_range() 153
fl_get_input_topline() 154
fl_get_input_xoffset() 154
fl_get_label_char_at_mouse() 291
fl_get_linestyle(() 263
fl_get_linewidth() 263
fl_get_matrix() 327
fl_get_menu() 225
fl_get_menu_item_mode() 226
fl_get_menu_item_text() 225
fl_get_menu_maxitems() 225
fl_get_menu_popup() 227
fl_get_menu_text() 225
fl_get_mouse() 50, 256
fl_get_nmenu_item() 166
fl_get_nmenu_item_by_label() 167
fl_get_nmenu_item_by_text() 167
fl_get_nmenu_item_by_value() 167
fl_get_nmenu_popup() 168
fl_get_object_bbox() 258, 287
fl_get_object_boxtype() 287
fl_get_object_bw() 26, 287
fl_get_object_color() 23, 287
fl_get_object_component() 288
fl_get_object_dblclick() 289
fl_get_object_geometry() 287
fl_get_object_gravity() 41, 289
fl_get_object_label() 31, 288
fl_get_object_lalign() 288
fl_get_object_lcolor() 288
fl_get_object_lsize() 288
fl_get_object_lstyle() 288
fl_get_object_objclass() 286
fl_get_object_position() 287
fl_get_object_resize() 41, 289
fl_get_object_return_state() 45, 176
fl_get_object_size() 287
fl_get_object_type() 286
fl_get_pattern() 77
fl_get_pixel() 255
fl_get_pixmap_pixmap() 115
fl_get_pixmapbutton_pixmap() 125
fl_get_positioner_mouse_buttons() 137
fl_get_positioner_numb() 137
fl_get_positioner_xbounds() 138
fl_get_positioner_xvalue() 138
fl_get_positioner_ybounds() 138
fl_get_positioner_yvalue() 138
fl_get_real_object_window() 242
fl_get_resource() 312
fl_get_scrollbar_bounds() 133
fl_get_scrollbar_increment() 133
fl_get_scrollbar_size() 133
fl_get_scrollbar_value() 133
fl_get_select_item() 160
fl_get_select_item_by_label() 160
fl_get_select_item_by_label_f() 160
fl_get_select_item_by_text() 160
fl_get_select_item_by_text_f() 160
fl_get_select_item_by_value() 160
fl_get_select_popup() 162
fl_get_select_text_align() 162
fl_get_select_text_color() 162
fl_get_select_text_font() 162
fl_get_slider_bounds() 129
fl_get_slider_repeat() 130
fl_get_slider_size() 130
fl_get_slider_value() 20, 129
fl_get_spinner_bounds() 143
fl_get_spinner_down_button() 143
fl_get_spinner_input() 143
fl_get_spinner_precision() 143
fl_get_spinner_step() 143
fl_get_spinner_up_button() 143
fl_get_spinner_value() 143
fl_get_string_dimension() 258
fl_get_string_height() 258
fl_get_string_width() 258

fl_get_tabfolder_folder_byname() 180
fl_get_tabfolder_folder_byname_f() 180
fl_get_tabfolder_folder_bynumber() 180
fl_get_tabfolder_numfolders() 180
fl_get_tabfolder_offset() 180
fl_get_text_clipping(..... 257
fl_get_thumbwheel_bounds() 145
fl_get_thumbwheel_step() 146
fl_get_thumbwheel_value() 145
fl_get_timer() 187
fl_get_vclass() 254
fl_get_win_mouse() 50, 256
fl_get_wingeometry() 50, 305
fl_get_winorigin() 50, 305
fl_get_winsize() 50, 305
fl_get_xyplot() 189
fl_get_xyplot_data() 190
fl_get_xyplot_data_pointer() 193
fl_get_xyplot_data_size() 190
fl_get_xyplot_numdata() 194
fl_get_xyplot_overlay_data() 193
fl_get_xyplot_screen_area() 196
fl_get_xyplot_world_area() 196
fl_get_xyplot_xbounds() 193
fl_get_xyplot_xmapping() 195
fl_get_xyplot_ybounds() 193
fl_get_xyplot_ymapping() 195
fl_getmcolor() 25, 256
fl_getpup_items() 235
fl_getpup_mode() 234
fl_getpup_text() 234
fl_gettime() 302
fl_hide_alert() 69
fl_hide_choice() 71
fl_hide_command_log() 74
fl_hide_form() 42, 295
fl_hide_fselector() 76
fl_hide_input() 71
fl_hide_message() 69
fl_hide_object() 22, 289
fl_hide_oneliner() 69
fl_hide_question() 70
fl_hidepup() 236
fl_iconify() 305
fl_initial_wingeometry() 304
fl_initial_winsize() 303
fl_initial_winstate() 303
fl_initialize() 277
fl_input_end_return_handling() 150
fl_insert_browser_line() 171
fl_insert_browser_line_f() 171
fl_insert_chart_value() 118
fl_insert_formbrowser() 184
fl_insert_nmenu_items() 167
fl_insert_nmenu_items2() 167
fl_insert_select_items() 161
fl_insert_xyplot_data() 194
fl_interpolate() 196
fl_invalidate_fselector_cache() 76
fl_is_center_lalign() 30
fl_is_clipped() 257
fl_is_global_clipped() 257
fl_is_inside_lalign() 30
fl_is_outside_lalign() 30
fl_is_text_clipped(..... 257
fl_isselected_browser_line() 173
fl_last_event() 50, 290
fl_library_version() 277
fl_line() 261
fl_lines() 261
fl_linestyle() 261
fl_linewidth() 261
fl_load_browser() 172
fl_lower_form() 294
fl_make_matrix() 347
fl_make_object() 252
fl_malloc() 241
fl_mapcolor() 25, 256
fl_mapcolorname() 25, 256
fl_mode_capable() 302
fl_mouse_button() 50
fl_move_object() 287
fl_msleep() 302
fl_newpup() 227
fl_noborder() 304
fl_now() 302
fl_object_is_active() 23, 297
fl_object_is_automatic() 289
fl_object_is_visible() 22, 290
fl_object_ps_dump() 291
fl_oval() 260
fl_ovalarc(..... 261
fl_ovalbound() 260
fl_ovalf() 260
fl_ovall() 260
fl_pclose() 73
fl_pieslice() 261
fl_point() 261
fl_points() 261
fl_polybound() 259
fl_polyf() 259
fl_polyl() 259
fl_popen() 73
fl_popup_add() 202
fl_popup_add_entries() 202
fl_popup_add_items() 211
fl_popup_create() 209
fl_popup_entry_clear_state() 214
fl_popup_entry_delete() 208
fl_popup_entry_get_by_label() 215
fl_popup_entry_get_by_label_f() 215
fl_popup_entry_get_by_position() 215
fl_popup_entry_get_by_text() 214
fl_popup_entry_get_by_text_f() 214
fl_popup_entry_get_by_user_data() 215
fl_popup_entry_get_by_value() 215

fl_popup_entry_get_group() 217
fl_popup_entry_get_state() 214
fl_popup_entry_get_subpopup() 218
fl_popup_entry_set_callback() 213
fl_popup_entry_set_enter_callback() 213
fl_popup_entry_set_font() 216
fl_popup_entry_set_group() 218
fl_popup_entry_set_leave_callback() 213
fl_popup_entry_set_shortcut() 217
fl_popup_entry_set_state() 214
fl_popup_entry_set_subpopup() 218
fl_popup_entry_set_text() 217
fl_popup_entry_set_user_data() 217
fl_popup_entry_set_value() 217
fl_popup_get_bw() 216
fl_popup_get_color() 216
fl_popup_get_min_width() 216
fl_popup_get_policy() 212
fl_popup_get_size() 213
fl_popup_get_title() 215
fl_popup_get_title_font() 215
fl_popup_insert_items() 211
fl_popup_raise_clear_state() 214
fl_popup_set_bw() 216
fl_popup_set_callback() 213
fl_popup_set_color() 216
fl_popup_set_cursor() 216
fl_popup_set_min_width() 216
fl_popup_set_policy() 212
fl_popup_set_position() 213
fl_popup_set_title() 215
fl_popup_set_title_f() 215
fl_popup_set_title_font() 215
fl_popup_toggle_clear_state() 214
fl_prepare_form_window() 40, 294
fl_prepare_form_window_f() 294
fl_print_xevent_name() 51
fl_raise_form() 294
fl_read_bitmapfile() 113
fl_read_pixmapfile() 115
fl_realloc() 241
fl_rect() 259
fl_rectbound() 259
fl_rectf() 259
fl_redraw_form() 32, 297
fl_redraw_object() 32, 297
fl_refresh_fselector() 77
fl_register_raw_callback() 314
fl_remove_canvas_handler() 198
fl_remove_event_callback() 53, 297
fl_remove_fselector_appbutton() 77
fl_remove_io_callback() 55
fl_remove_selected_xevent() 53
fl_remove_signal_callback() 298
fl_remove_timeout() 47, 300
fl_replace_browser_line() 172
fl_replace_browser_line_f() 172
fl_replace_chart_value() 119
fl_replace_choice() 220
fl_replace_choice_f() 220
fl_replace_formbrowser() 184
fl_replace_menu_item() 224
fl_replace_nmenu_item() 167
fl_replace_nmenu_items2() 167
fl_replace_select_item() 161
fl_replace_xyplot_point() 193
fl_request_clipboard() 307
fl_reset_cursor() 306
fl_reset_focus_object() 290
fl_reset_winconstraints() 305
fl_resume_timer() 187
fl_ringbell() 302
fl_roundrect() 259
fl_roundrectf() 259
fl_scale_form() 40, 294
fl_select_browser_line() 173
fl_set_app_mainform() 39, 294
fl_set_app_nomainform() 294
fl_set_atclose() 43
fl_set_background() 255
fl_set_bitmap_data() 113
fl_set_bitmap_file() 113
fl_set_bitmapbutton_data() 125
fl_set_bitmapbutton_file() 125
fl_set_border_width() 27, 281
fl_set_browser_bottomline() 174
fl_set_browser_dblclick_callback() 173
fl_set_browser_fontsize() 174
fl_set_browser_fontstyle() 174
fl_set_browser_hscroll_callback() 176
fl_set_browser_hscrollbar() 175
fl_set_browser_rel_xoffset() 174
fl_set_browser_rel_yoffset() 174
fl_set_browser_scrollbar_repeat() 177
fl_set_browser_scrollbarsize() 176
fl_set_browser_specialkey() 175
fl_set_browser_topline() 174
fl_set_browser_vscroll_callback() 176
fl_set_browser_vscrollbar() 175
fl_set_browser_xoffset() 174
fl_set_browser_yoffset() 174
fl_set_button() 18, 124
fl_set_button_mouse_buttons() 124
fl_set_button_shortcut() 124
fl_set_canvas_attributes() 199
fl_set_canvas_colormap() 199
fl_set_canvas_depth() 199
fl_set_canvas_visual() 199
fl_set_chart_autosize() 119
fl_set_chart_baseline() 119
fl_set_chart_bounds() 119
fl_set_chart_lcolor() 118
fl_set_chart_lsize() 118
fl_set_chart_lstyle() 118
fl_set_chart_maxnumb() 119
fl_set_choice() 221

fl_set_choice_align() 221
fl_set_choice_align_bottom() 222
fl_set_choice_entries() 221
fl_set_choice_fontsize() 221
fl_set_choice_fontstyle() 222
fl_set_choice_item_mode() 221
fl_set_choice_text() 221
fl_set_choice_text_f() 221
fl_set_choices_shortcut() 71
fl_set_clipping() 257
fl_set_clock_adjustment() 117
fl_set_clock_ampm() 117
fl_set_color_leak() 26
fl_set_command_log_position() 74
fl_set_coordunit() 280
fl_set_counter_bounds() 140
fl_set_counter_filter() 140
fl_set_counter_min_repeat() 141
fl_set_counter_precision() 140
fl_set_counter_repeat() 141
fl_set_counter_speedjump() 141
fl_set_counter_step() 140
fl_set_counter_value() 140
fl_set_cursor() 306
fl_set_cursor_color() 306
fl_set_default_tabfolder_corner() 181
fl_set_defaults() 279
fl_set_dial_angles() 135
fl_set_dial_bounds() 135
fl_set_dial_crossover() 135
fl_set_dial_direction() 135
fl_set_dial_step() 135
fl_set_dial_value() 135
fl_set_directory() 77
fl_set_dirlist_filter() 80
fl_set_dirlist_filterdir() 80
fl_set_dirlist_sort() 80
fl_set_error_handler() 282
fl_set_error_logfp() 283
fl_set_event_callback() 52, 297
fl_set_focus_object() 21, 290
fl_set_folder() 181
fl_set_folder_byname() 181
fl_set_folder_byname_f() 181
fl_set_folder_bynumber() 181
fl_set_font_name() 29, 283
fl_set_font_name_f() 29, 283
fl_set_foreground() 255
fl_set_form_atactivate() 48, 296
fl_set_form_atclose() 43
fl_set_form_atdeactivate() 48, 296
fl_set_form_background_color() 285
fl_set_form_callback() 52, 290
fl_set_form_dblbuffer() 282
fl_set_form_geometry() 38, 294
fl_set_form_hotobject() 39
fl_set_form_hotspot() 39
fl_set_form_icon() 42, 295
fl_set_form_maxsize() 295
fl_set_form_minsize() 295
fl_set_form_position() 38, 294
fl_set_form_size() 38, 294
fl_set_form_title() 42, 295
fl_set_form_title_f() 42, 295
fl_set_formbrowser_hscrollbar() 184
fl_set_formbrowser_scroll() 184
fl_set_formbrowser_topform() 185
fl_set_formbrowser_topform_bynumber() 185
fl_set_formbrowser_vscrollbar() 184
fl_set_formbrowser_xoffset() 185
fl_set_formbrowser_yoffset() 185
fl_set_fselector_border() 77
fl_set_fselector_callback() 76
fl_set_fselector_filetype_marker() 78
fl_set_fselector_fontsize() 76
fl_set_fselector_fontstyle() 76
fl_set_fselector_placement() 77
fl_set_fselector_title() 76
fl_set_gc_clipping() 258
fl_set_glcanvas_attributes() 201
fl_set_glcanvas_defaults() 200
fl_set_glcanvas_direct() 201
fl_set_goodies_font() 71
fl_set_icm_color() 25, 284
fl_set_idle_callback() 46, 300
fl_set_idle_delta() 300
fl_set_input() 21, 152
fl_set_input_color() 21, 154
fl_set_input_cursor_visible() 154
fl_set_input_cursorpos() 153
fl_set_input_editkeymap() 155
fl_set_input_f() 21, 152
fl_set_input_fieldchar() 154
fl_set_input_filter() 150
fl_set_input_format() 151
fl_set_input_hscrollbar() 153
fl_set_input_maxchars() 150
fl_set_input_mode() 152
fl_set_input_return() 149
fl_set_input_scroll() 153
fl_set_input_scrollbarsize() 154
fl_set_input_selected() 152
fl_set_input_selected_range() 152
fl_set_input_topline() 154
fl_set_input_vscrollbar() 153
fl_set_input_xoffset() 154
fl_set_menu() 223
fl_set_menu_entries() 224
fl_set_menu_item_callback() 226
fl_set_menu_item_mode() 225
fl_set_menu_item_shortcut() 226
fl_set_menu_notitle() 227
fl_set_menu_popup() 227
fl_set_mouse() 257
fl_set_nmenu_hl_text_color() 168
fl_set_nmenu_items() 164

fl_set_nmenu_policy() 166
fl_set_nmenu_popup() 165
fl_set_object_automatic() 289
fl_set_object_boxtype() 26, 287
fl_set_object_bw() 26, 287
fl_set_object_callback() 51, 290
fl_set_object_color() 23, 287
fl_set_object_dblbuffer() 282
fl_set_object_dblclick() 289
fl_set_object_geometry() 287
fl_set_object_gravity() 40, 289
fl_set_object_helper() 31, 288
fl_set_object_helper_f() 31, 288
fl_set_object_label() 31, 288
fl_set_object_label_f() 31, 288
fl_set_object_lalign() 30, 288
fl_set_object_lcolor() 27, 288
fl_set_object_lsize() 27, 288
fl_set_object_lstyle() 28, 288
fl_set_object_position() 287
fl_set_object_posthandler() 275, 315
fl_set_object_prehandler() 275, 315
fl_set_object_resize() 40, 289
fl_set_object_return() 44
fl_set_object_shortcut() 245
fl_set_object_shortcutkey() 245
fl_set_object_size() 287
fl_set_oneliner_color() 69
fl_set_oneliner_font() 69
fl_set_pattern() 77
fl_set_pixmap_align() 115
fl_set_pixmap_colorcloseness() 115
fl_set_pixmap_data() 114
fl_set_pixmap_file() 114
fl_set_pixmap_pixmap() 115
fl_set_pixmapbutton_align() 125
fl_set_pixmapbutton_data() 125
fl_set_pixmapbutton_file() 125
fl_set_pixmapbutton_focus_data() 126
fl_set_pixmapbutton_focus_file() 126
fl_set_pixmapbutton_focus_outline() 126
fl_set_pixmapbutton_focus_pixmap() 126
fl_set_pixmapbutton_pixmap() 125
fl_set_positioner_mouse_buttons() 137
fl_set_positioner_xbounds() 137
fl_set_positioner_xstep() 138
fl_set_positioner_xvalue() 137
fl_set_positioner_ybounds() 137
fl_set_positioner_ystep() 138
fl_set_positioner_yvalue() 137
fl_set_resource() 72, 312
fl_set_scrollbar_bounds() 133
fl_set_scrollbar_increment() 133
fl_set_scrollbar_size() 133
fl_set_scrollbar_type() 281
fl_set_scrollbar_value() 133
fl_set_select_item() 161
fl_set_select_items() 158
fl_set_select_policy() 160
fl_set_select_popup() 159
fl_set_select_text_align() 162
fl_set_select_text_color() 162
fl_set_select_text_font() 162
fl_set_slider_bounds() 20, 129
fl_set_slider_filter() 130
fl_set_slider_precision() 130
fl_set_slider_repeat() 130
fl_set_slider_size() 130
fl_set_slider_value() 20, 129
fl_set_spinner_bounds() 143
fl_set_spinner_precision() 143
fl_set_spinner_step() 143
fl_set_spinner_value() 143
fl_set_tabfolder_autofit() 181
fl_set_tabfolder_offset() 180
fl_set_tabstop() 282
fl_set_text_clipping() 257
fl_set_thumbwheel_bounds() 145
fl_set_thumbwheel_crossover() 145
fl_set_thumbwheel_step() 146
fl_set_thumbwheel_value() 145
fl_set_timer() 186
fl_set_timer_countup() 187
fl_set_timer_filter() 187
fl_set_tooltip_boxtype() 31, 288
fl_set_tooltip_color() 31, 288
fl_set_tooltip_font() 31, 288
fl_set_tooltip_lalign() 31
fl_set_visualID() 279
fl_set_xyplot_alphaxtics() 191
fl_set_xyplot_alphaytics() 191
fl_set_xyplot_data() 189
fl_set_xyplot_data_double() 189
fl_set_xyplot_file() 190
fl_set_xyplot_fixed_xaxis() 192
fl_set_xyplot_fixed_yaxis() 192
fl_set_xyplot_grid_linestyle() 191
fl_set_xyplot_inspect() 190
fl_set_xyplot_interpolate() 195
fl_set_xyplot_key() 194
fl_set_xyplot_key_font() 195
fl_set_xyplot_key_position() 194
fl_set_xyplot_keys() 195
fl_set_xyplot_linewidth() 195
fl_set_xyplot_log_minor_xtics() 191
fl_set_xyplot_log_minor_ytics() 191
fl_set_xyplot_mark_active() 189
fl_set_xyplot_maxoverlays() 194
fl_set_xyplot_overlay_type() 193
fl_set_xyplot_symbol() 192
fl_set_xyplot_symbolsize() 192
fl_set_xyplot_xbounds() 193
fl_set_xyplot_xgrid() 191
fl_set_xyplot_xscale() 195
fl_set_xyplot_xtics() 190
fl_set_xyplot_ybounds() 193

fl_set_xyplot_ygrid() 191
fl_set_xyplot_yscale() 195
fl_set_xyplot_ytics() 190
fl_setpup_align_bottom() 234
fl_setpup_bw() 235
fl_setpup_cursor() 235
fl_setpup_default_bw(); 235
fl_setpup_default_checkcolor() 236
fl_setpup_default_color() 236
fl_setpup_default_cursor() 235
fl_setpup_default_fontsize() 227, 235
fl_setpup_default_fontstyle() 227, 235
fl_setpup_entercb() 233
fl_setpup_entries() 231
fl_setpup_itemcb() 233
fl_setpup_leavecb() 233
fl_setpup_maxpups() 236
fl_setpup_menucb() 233
fl_setpup_mode() 234
fl_setpup_position() 234
fl_setpup_selection() 235
fl_setpup_shadow() 235
fl_setpup_shortcut() 233
fl_setpup_softedge() 235
fl_setpup_submenu() 234
fl_setpup_title() 235
fl_setpup_title_f() 235
fl_share_canvas_colormap() 199
fl_show_alert() 69
fl_show_alert_f() 70
fl_show_browser_line() 173
fl_show_choice() 71
fl_show_choices() 71
fl_show_colormap() 75
fl_show_command_log() 74
fl_show_errors() 283
fl_show_form() 37, 292
fl_show_form_f() 292
fl_show_form_window() 40, 294
fl_show_fselector() 75
fl_show_input() 71
fl_show_menu_symbol() 226
fl_show_message() 69
fl_show_messages() 69
fl_show_messages_f() 69
fl_show_object() 22, 289
fl_show_oneliner() 69
fl_show_question() 70
fl_show_simple_input() 72
fl_showpup() 236
fl_signal_caught() 299
fl_strdup() 241
fl_stuff_clipboard() 307
fl_suspend_timer() 187
fl_to_inside_lalign() 30
fl_to_outside_lalign() 30
fl_transient() 304
fl_trigger_object() 23, 290
fl_unfreeze_all_forms() 289
fl_unfreeze_form() 32, 289
fl_unset_clipping() 257
fl_unset_gc_clipping() 258
fl_unset_text_clipping() 257
fl_update_display() 37
fl_use_fselector() 76
fl_validate_input() 151
fl_whoami() 302
fl_win_to_form() 255
fl_winaspect() 304
fl_winbackground() 304
fl_winclose() 305
fl_wincreate() 302
fl_winfocus() 306
fl_wingeometry() 304
fl_winget() 259
fl_winhide() 305
fl_winicon() 304
fl_winicontitle() 304
fl_winicontitle_f() 304
fl_winisvalid() 305
fl_winmaxsize() 303
fl_winminsize() 303
fl_winmove() 305
fl_winopen() 303
fl_winposition() 304
fl_winreparent() 303
fl_winreshape() 305
fl_winresize() 305
fl_winset() 259
fl_winshow() 302
fl_winsize() 303
fl_winstepsize() 305
fl_wintitle() 304
fl_wintitle_f() 304
fl_XEventsQueued() 49
fl_XNextEvent() 49
fl_XPeekEvent() 49
fl_XPutbackEvent() 49
fl_xyplot_s2w() 196
fl_xyplot_w2s() 196
FL_FormDisplay() 255
FL_ObjWin() 198
flimage_add_format() 330
flimage_add_marker() 345
flimage_add_marker_struct() 345
flimage_add_text() 344
flimage_add_text_struct() 344
flimage_alloc() 326
flimage_autocrop() 340
flimage_close() 338
flimage_convert() 326
flimage_convolve() 338
flimage_convolvea() 339
flimage_crop() 340
flimage_define_marker() 346
flimage_delete_all_markers() 346

flimage_delete_all_text() 345
flimage_description_via_filter() 333
flimage_display() 320
flimage_dump() 319
flimage_dup() 348
flimage_enable_bmp() 329
flimage_enable_fits() 329
flimage_enable_genesis() 329
flimage_enable_gif() 329
flimage_enable_gzip() 329
flimage_enable_jpeg() 329
flimage_enable_png() 329
flimage_enable_pnm() 329
flimage_enable_ps() 329
flimage_enable_sgi() 329
flimage_enable_tiff() 329
flimage_enable_xbm() 329
flimage_enable_xpm() 329
flimage_enable_xwd() 329
flimage_error() 337
flimage_flip() 340
flimage_free() 320
flimage_from_pixmap() 348
flimage_get_autocrop() 341
flimage_get_format_info() 335
flimage_get_number_of_formats() 335
flimage_getmem() 327
flimage_gif_output_options() 329
flimage_is_supported() 321
flimage_jpeg_output_options() 329
flimage_load() 319
flimage_open() 338
flimage_pnm_output_options() 329
flimage_ps_options() 329
flimage_read() 338
flimage_render_annotation() 347
flimage_rotate() 339
flimage_scale() 341
flimage_sdisplay() 320
flimage_select_mediancut_quantizer() 349
flimage_select_octree_quantizer() 349
flimage_setup() 336
flimage_tint() 339
flimage_to_pixmap() 348
flimage_transform_pixels() 343
flimage_type_name() 325
flimage_warp() 342
flimage_windowlevel() 325
flimage_write_via_filter() 333
flps_init() 291

Index of Global Variables

fl_current_form 300
fl_display 301
fl_root 301
fl_screen 301
fl_scrh 301
fl_scrw 301
fl_state 301
fl_ul_magic_char 301
fl_vmode 301
fl_vroot 301
FL_EVENT 300

Index of Constants

FD_FSELECTOR 78
fl_popup_delete() 208
fl_popup_do() 211
fl_popup_insert_entries() 208
FL DASH 262
FL DOT 261
FL DOTDASH 262
FL LONGDASH 262
FL PUP NONE 231
FL SOLID 261
FL USERDASH 262
FL USERDOUBLEDASH 262
FL_ACTIVE_XYPLOT 188
FL_ALIGN_BOTTOM 30
FL_ALIGN_CENTER 30
FL_ALIGN_INSIDE 30
FL_ALIGN_LEFT 30
FL_ALIGN_LEFT_BOTTOM 30
FL_ALIGN_LEFT_TOP 30
FL_ALIGN_RIGHT 30
FL_ALIGN_RIGHT_BOTTOM 30
FL_ALIGN_RIGHT_TOP 30
FL_ALIGN_TOP 30
FL_ALL_FREE 58
FL_ALPHASORT 80
FL_ALT_MASK 156
FL_ANALOG_CLOCK 116
FL_APPEVENT_CB 46, 52, 297, 300
FL_ATTRIB 244
FL_AUTO 153, 176, 184
FL_BAR_CHART 117
FL_BEGIN_GROUP 241, 285
FL_BEING_HIDDEN 42
FL_BITMAPBUTTON 121
FL_BLACK 24
FL_BLUE 24
FL_BOLD_STYLE 28
FL_BOLDITALIC_STYLE 28
FL_BOOL 309
FL_BORDER_BOX 15, 109
FL_BORDER_FRAME 110, 111
FL_BOTTOM_BCOL 24
FL_BOTTOM_TABFOLDER 178
FL_BROWSER_SCROLL_CALLBACK 176
FL_BUTTON 121
FL_BUTTON_NMENU 163
FL_BUTTON_STRUCT 268
FL_BUTTON_TOUCH_NMENU 163
FL_CALLBACKPTR 51
FL_CASEALPHASORT 80
FL_CHART_MAX 119
FL_CHARTREUSE 24
FL_CHECKBUTTON 121
FL_CIRCLE_XYPLOT 188
FL_CLICK_TIMEOUT 243
FL_CMD_OPT 309
FL_COL1 24
FL_CONTINUOUS_FREE 58
FL_CONTROL_MASK 156
FL_COORD_centiMM 281
FL_COORD_centiPOINT 281
FL_COORD_MM 281
FL_COORD_PIXEL 281
FL_COORD_POINT 281
FL_CYAN 24
FL_DARKCYAN 24
FL_DARKER_COL1 24
FL_DARKGOLD 24
FL_DARKORANGE 24
FL_DARKTOMATO 24
FL_DARKVIOLET 24
FL_DASHED_XYPLOT 188
FL_DATE_INPUT 147
FL_DBLCLICK 57, 243
FL_DEEPPINK 24
FL_DESELECTABLE_HOLD_BROWSER 169
FL_DIAL_CCW 135
FL_DIAL_CW 135
FL_DIGITAL_CLOCK 116
FL_Dirlist 79
FL_DIRLIST_FILTER 80
FL_DODGERBLUE 25
FL_DOS_INPUT_MODE 152
FL_DOTDASHED_XYPLOT 188
FL_DOTTED_XYPLOT 188
FL_DOWN_BOX 15, 109
FL_DOWN_FRAME 110, 111
FL_DRAW 56, 242
FL_DRAWLABEL 57, 242
FL_DROPLIST_CHOICE 219
FL_DROPLIST_SELECT 157
FL_East 41
FL_EditKeymap 155
FL_EMBOSSED_BOX 16
FL_EMBOSSED_FRAME 110, 112
FL_EMBOSSED_STYLE 28
FL_EMPTY_XYPLOT 188
FL_END_GROUP 285
FL_ENGRAVED_FRAME 110, 111
FL_ENGRAVED_STYLE 28
FL_ENLARGE_ONLY 181
FL_ENTER 57, 243
FL_EXCEPT 55
FL_FILL_DIAL 134
FL_FILL_XYPLOT 188
FL_FILLED_CHART 118
FL_FIT 181
FL_FIXED_STYLE 28
FL_FIXEDBOLD_STYLE 28
FL_FIXEDBOLDITALIC_STYLE 28

FL_FIXEDITALIC_STYLE 28
FL_FLAT_BOX 16, 109
FL_FLOAT 310
FL_FLOAT_INPUT 147
FL_FLOAT_SPINNER 142
FL_FOCUS 57, 244
FL_FORM_ATACTIVATE 48, 296
FL_FORM_ATDEACTIVATE 48, 296
FL_FRAME_BOX 16, 109
FL_FREE_COL1 25
FL_FREE_SIZE 38
FL_FREEMEM 58, 244
FL_FULLBORDER 39, 293
FL_GREEN 24
FL_GRID_MAJOR 191
FL_GRID_MINOR 191
FL_GRID_NONE 191
FL_HANDLE_CANVAS 197
FL_HANDLEPTR 252, 275, 315
FL_HIDDEN_BUTTON 122
FL_HIDDEN_INPUT 147
FL_HIDDEN_RET_BUTTON 122
FL_HIDDEN_TIMER 186
FL_HOLD_BROWSER 169
FL_HOR_BROWSER_SLIDER 127
FL_HOR_FILL_SLIDER 127
FL_HOR_NICE_SCROLLBAR 131
FL_HOR_NICE_SLIDER 127
FL_HOR_PLAIN_SCROLLBAR 132
FL_HOR_PROGRESS_BAR 127
FL_HOR_SCROLLBAR 131
FL_HOR_SLIDER 127
FL_HOR_THIN_SCROLLBAR 131
FL_HOR_THUMBWHEEL 144
FL_HORBAR_CHART 117
FL_HUGE_SIZE 27
FL_IMAGE 322
FL_IMAGE_CI 325
FL_IMAGE_FLEX 325
FL_IMAGE_GRAY 325
FL_IMAGE_GRAY16 325
FL_IMAGE_MONO 325
FL_IMAGE_PACKED 325
FL_IMAGE_RGB 325
FL_IMPULSE_XYPLOT 188
FL_INACTIVE 24
FL_INACTIVE_FREE 58
FL_INCLUDE_VERSION 277
FL_INDIANRED 24
FL_INOUT_BUTTON 122
FL_INPUT_DDMM 151
FL_INPUT_END_EVENT_ALWAYS 150
FL_INPUT_END_EVENT_CLASSIC 150
FL_INPUT_FREE 58
FL_INPUT_MMDD 151
FL_INPUTVALIDATOR 150
FL_INT 309
FL_INT_INPUT 147
FL_INT_SPINNER 142
FL_INVALID 151
FL_INVISIBLE 42
FL_INVISIBLE_CURSOR 306
FL_INVISIBLE_POSITIONER 136
FL_IO_CALLBACK 54
FL_IOPT 279
FL_ITALIC_STYLE 28
FL_JUMP_SCROLL 184
FL_KEY_ALL 246
FL_KEY_NORMAL 246
FL_KEY_SPECIAL 246
FL_KEY_TAB 246
FL_KEYPRESS 58, 244
FL_LABELBUTTON 121
FL_LARGE_SIZE 27
FL_LEAVE 57, 243
FL_LEFT_BCOL 24
FL_LEFT_MOUSE 243
FL_LIGHTBUTTON 121
FL_LIGHTER_COL1 24
FL_LINE_CHART 117
FL_LINE_DIAL 134
FL_LINEAR 195
FL_LINEPOINTS_XYPLOT 188
FL_LOG 195
FL_LONG 309
FL_LOSE_SELECTION_CB 307
FL_MAGENTA 24
FL_MAX_COLORS 25
FL_MAX_FSELECTOR 76
FL_MAX_XYPLOTOVERLAY 193
FL_MAXPUPI 229
FL_MBUTTON1 243
FL_MBUTTON2 243
FL_MBUTTON3 243
FL_MBUTTON4 243
FL_MBUTTON5 243
FL_MCOL 24
FL_MEDIUM_SIZE 27
FL_MENU_BUTTON 122
FL_MENU_SELECT 157
FL_MIDDLE_MOUSE 243
FL_MOTION 57, 243
FL_MTIMESORT 80
FL_MULTI_BROWSER 169
FL_MULTILINE_INPUT 147
FL_NO 181
FL_NO_BOX 15, 109
FL_NO_FRAME 110, 111
FL_NOBORDER 39, 293
FL_NOEVENT 290
FL_NoGravity 40
FL_NONE 80
FL_NORMAL_BITMAP 113
FL_NORMAL_BROWSER 169
FL_NORMAL_BUTTON 18, 122
FL_NORMAL_CANVAS 197

FL_NORMAL_CHOICE 219
FL_NORMAL_CHOICE2 219
FL_NORMAL_COUNTER 139
FL_NORMAL_DIAL 134
FL_NORMAL_FORMBROWSER 182
FL_NORMAL_FREE 58
FL_NORMAL_INPUT 147
FL_NORMAL_INPUT_MODE 152
FL_NORMAL_NMENU 163
FL_NORMAL_PIXMAP 114
FL_NORMAL_POSITIONER 136
FL_NORMAL_SELECT 157
FL_NORMAL_SIZE 27
FL_NORMAL_STYLE 28
FL_NORMAL_TEXT 112
FL_NORMAL_TIMER 186
FL_NORMAL_TOUCH_NMENU 163
FL_NORMAL_XYPLOT 188
FL_North 41
FL_NorthEast 41
FL_NorthWest 40
FL_OFF 153, 176, 184
FL_ON 153, 176, 184
FL_ORCHID 24
FL_OTHER 58, 244
FL_OVAL_BOX 16, 109
FL_OVAL_FRAME 110, 112
FL_OVAL3D_DOWNBOX 16
FL_OVAL3D_UPBOX 16
FL_OVERLAY_POSITIONER 136
FL_PACKED 322
FL_PALEGREEN 24
FL_PCBITS 322
FL_PCMAX 322
FL_PIE_CHART 118
FL_PIXMAPBUTTON 121
FL_PLACE GEOMETRY 292
FL_PLACE ICONIC 293
FL_PLACE_ASPECT 37, 292
FL_PLACE_CENTER 37, 293
FL_PLACE_CENTERFREE 38, 293
FL_PLACE_FREE 38, 293
FL_PLACE_FULLSCREEN 38, 293
FL_PLACE_GEOMETRY 37
FL_PLACE_HOTSPOT 38, 293
FL_PLACE_ICONIC 38
FL_PLACE_MOUSE 37, 292
FL_PLACE_POSITION 37, 292
FL_PLACE_SIZE 37, 292
FL_POINT 259
FL_POINTS_XYPLOT 188
FL_POPUP 207
FL_POPUP_BACKGROUND_COLOR 217
FL_POPUP_CB 205
FL_POPUP_CHECKED 165, 207, 214
FL_POPUP_DISABLED 207, 214
FL_POPUP_DISABLED_TEXT_COLOR 217
FL_POPUP_DRAG_SELECT 160, 166, 212
FL_POPUP_DSABLED 164
FL_POPUP_ENTRY 206
FL_POPUP_HIDDEN 165, 207, 214
FL_POPUP_HIGHLIGHT_COLOR 217
FL_POPUP_HIGHLIGHT_TEXT_COLOR 217
FL_POPUP_ITEM 209
FL_POPUP_LINE 207
FL_POPUP_NONE 164, 207
FL_POPUP_NORMAL 164, 207
FL_POPUP_NORMAL_SELECT 160, 166, 212
FL_POPUP_RADIO 164, 207
FL_POPUP_RADIO_COLOR 217
FL_POPUP_RETURN 206
FL_POPUP_SUB 207
FL_POPUP_TEXT_COLOR 217
FL_POPUP_TITLE_COLOR 217
FL_POPUP_TOGGLE 164, 207
FL_PREEMPT 275, 314
FL_PULLDOWN_MENU 222
FL_PUP_BOX 226, 231
FL_PUP_CB 233
FL_PUP_CHECK 226, 232
FL_PUP_ENTERCB 233
FL_PUP_ENTRY 224, 231
FL_PUP_GREY 226, 231
FL_PUP_LEAVECB 233
FL_PUP_NONE 226
FL_PUP_RADIO 226, 232
FL_PUSH 57, 243
FL_PUSH_BUTTON 18, 122
FL_PUSH_MENU 222
FL_RADIO_BUTTON 18, 122
FL_RALPHASORT 80
FL_RAW_CALLBACK 314
FL_RCASEALPHASORT 80
FL_READ 55
FL_RED 24
FL_RELEASE 57, 243
FL_RESIZE_ALL 40
FL_RESIZE_NONE 40
FL_RESIZE_X 40
FL_RESIZE_Y 40
FL_resource 309
FL_RETURN_ALWAYS 45
FL_RETURN_BUTTON 122
FL_RETURN_CHANGED 44
FL_RETURN_DESELECTION 45
FL_RETURN_END 44
FL_RETURN_END_CHANGED 45
FL_RETURN_NONE 45
FL_RETURN_SELECTION 45
FL_REVISION 277
FL_RFLAT_BOX 16, 109
FL_RIGHT_BCOL 24
FL_RIGHT_MOUSE 243
FL_RINGBELL 151
FL_RMTIMESORT 80
FL_ROUND3DBUTTON 121

FL_ROUNDBUTTON ... 121
FL_ROUNDED_BOX ... 16, 109
FL_ROUNDED_FRAME ... 110, 112
FL_ROUNDED3D_DOWNBOX ... 16
FL_ROUNDED3D_UPBOX ... 16
FL_RSHADOW_BOX ... 16, 109
FL_RSIZESORT ... 80
FL_RTYPE ... 309
FL_SCROLLBARBUTTON ... 121
FL_SCROLLDOWN_MOUSE ... 243
FL_SCROLLUP_MOUSE ... 243
FL_SECRET_INPUT ... 147
FL_SELECT_BROWSER ... 169
FL_SELECTION_CB ... 307
FL_SHADOW_BOX ... 16, 109
FL_SHADOW_STYLE ... 28
FL_SHORT ... 309
FL_SHORTCUT ... 50, 58, 124, 244
FL_SIGNAL_HANDLER ... 298
FL_SIMPLE_COUNTER ... 139
FL_SIZESORT ... 80
FL_SLATEBLUE ... 24
FL_SLIDER_MAX_PREC ... 130
FL_SLIDER_WIDTH ... 130, 133
FL_SMALL_SIZE ... 27
FL_SMOOTH_SCROLL ... 184
FL_South ... 41
FL_SouthEast ... 41
FL_SouthWest ... 41
FL_SPECIALPIE_CHART ... 118
FL_SPIKE_CHART ... 118
FL_SPRINGGREEN ... 25
FL_SQUARE_XYPLOT ... 188
FL_State ... 254
FL_STEP ... 58, 244
FL_STRING ... 310
FL_TIMEOUT_CALLBACK ... 46, 300
FL_TIMER_FILTER ... 187
FL_TIMES_STYLE ... 28
FL_TIMESBOLD_STYLE FL ... 28
FL_TIMESBOLDITALIC_STYLE ... 28
FL_TIMESITALIC_STYLE ... 28
FL_TINY_SIZE ... 27
FL_TOMATO ... 24
FL_TOP_BCOL ... 24
FL_TOP_TABFOLDER ... 178
FL_TOUCH_BUTTON ... 18, 122
FL_TOUCH_MENU ... 222
FL_TRANSIENT ... 39, 293
FL_TRIGGER ... 290
FL_TRPLCLICK ... 57, 243
FL_UNFOCUS ... 58, 244
FL_UP_BOX ... 15, 109
FL_UP_FRAME ... 110, 111
FL_UPDATE ... 57, 244
FL_USER_CLASS_START ... 241
FL_VALID ... 151
FL_VALUE_TIMER ... 186
FL_VERSION ... 277
FL_VERT_BROWSER_SLIDER ... 127
FL_VERT_FILL_SLIDER ... 127
FL_VERT_NICE_SCROLLBAR ... 131
FL_VERT_NICE_SLIDER ... 127
FL_VERT_PLAIN_SCROLLBAR ... 132
FL_VERT_PROGRESS_BAR ... 127
FL_VERT_SCROLLBAR ... 131
FL_VERT_SLIDER ... 127
FL_VERT_THIN_SCROLLBAR ... 131
FL_VERT_THUMBWHEEL ... 144
FL_VISIBLE ... 42
FL_West ... 41
FL_WHEAT ... 24
FL_WHITE ... 24
FL_WRITE ... 55
FL_XYPLOT_SYMBOL ... 192
FL_YELLOW ... 24
FLIMAGE_ASPECT ... 341
FLIMAGE_AUTOCOLOR ... 340
FLIMAGE_CENTER ... 341
FLIMAGE_Description ... 330
FLIMAGE_FORMAT_INFO ... 335
FLIMAGE_Identify ... 330
FLIMAGE_JPEG_OPTIONS ... 329
FLIMAGE_NOCENTER ... 341, 343
FLIMAGE_NOSUBPIXEL ... 339, 341, 343
FLIMAGE_Read_Pixels ... 330
FLIMAGE_SETUP ... 336
FLIMAGE_SHARPEN ... 339
FLIMAGE_SMOOTH ... 339
FLIMAGE_SUBPIXEL ... 339, 341, 343
FLIMAGE_TEXT ... 344
FLIMAGE_Write_Image ... 330
FLPS_CONTROL ... 291
ForgetGravity ... 41
FT_BLK ... 79
FT_CHR ... 79
FT_DIR ... 79
FT_FIFO ... 79
FT_FILE ... 79
FT_LINK ... 79
FT_SOCK ... 79