FLTK 1.3.4 Programming Manual

Revision 9 by F. Costantini, D. Gibson, M. Melcher,
A. Schlosser, B. Spitzak, and M. Sweet.

Generated by Doxygen 1.8.6

November 11, 2016

Contents

Chapter 1

FLTK Programming Manual

	FLTK 1.3.4 Programming Manual Revision 9 by F. Costantini, D. Gibson, M. Melcher, A. Schlosser, B. Spitzak and M. Sweet. Copyright 1998-2016 by Bill Spitzak and others.

Chapter 2

Preface

This manual describes the Fast Light Tool Kit (”FLTK”) version 1.3.4, a C++ Graphical User Interface (”GUI”) toolkit for UNIX, Microsoft Windows and Apple OS X.

Each of the chapters in this manual is designed as a tutorial for using FLTK, while the appendices provide a convenient reference for all FLTK widgets, functions, and operating system interfaces.

This manual may be printed, modified, and/or used under the terms of the FLTK license provided in Software License.

2.1 Organization

This manual is organized into the following chapters and appendices:

- Developer Information
- Software License
- Example Source Code

2.2 Conventions

This manual was generated using Doxygen (see `http://www.doxygen.org/`) to process the source code itself, special comments in the code, and additional documentation files. In general, Doxygen recognizes and denotes the following entities as shown:

- classes, such as Fl_Widget,
- methods, such as Fl_Widget::callback(Fl_Callback* cb, void* p),
- functions, such as fl_draw(const char *str, int x, int y),
- internal links, such as Conventions,
- external links, such as `http://www.stack.nl/~dimitri/doxygen/`

Other code samples and commands are shown in `regular courier type`.

2.3 Abbreviations

The following abbreviations are used in this manual:

X11

The X Window System version 11.

Xlib

The X Window System interface library.

MS Windows, WIN32

The Microsoft Windows Application Programmer's Interface for Windows 2000, Windows XP, Windows Vista, and Windows 7. FLTK uses the preprocessor definition `WIN32` for the 32 bit and 64 bit MS Windows API.

OS X, __APPLE__

The Apple desktop operating sytem OS X 10.0 and later. MacOS 8 and 9 support was dropped after FLTK 1.0.10. FLTK uses the preprocessor definition `__APPLE__` for OS X.

2.4 Copyrights and Trademarks

FLTK is Copyright 1998-2016 by Bill Spitzak and others. Use and distribution of FLTK is governed by the GNU Library General Public License with 4 exceptions, located in Software License.

UNIX is a registered trademark of the X Open Group, Inc. Microsoft and Windows are registered trademarks of Microsoft Corporation. OpenGL is a registered trademark of Silicon Graphics, Inc. Apple, Macintosh, MacOS, and Mac OS X are registered trademarks of Apple Computer, Inc.

Chapter 3

Introduction to FLTK

The Fast Light Tool Kit (”FLTK”, pronounced ”fulltick”) is a cross-platform C++ GUI toolkit for UNI-X®/Linux® (X11), Microsoft® Windows®, and Apple® OS X®.

FLTK provides modern GUI functionality without the bloat and supports 3D graphics via OpenG-L® and its built-in GLUT emulation. It was originally developed by Mr. Bill Spitzak and is currently maintained by a small group of developers across the world with a central repository in the US.

3.1 History of FLTK

It has always been Bill’s belief that the GUI API of all modern systems is much too high level. Toolkits (even FLTK) are *not* what should be provided and documented as part of an operating system. The system only has to provide arbitrary shaped but featureless windows, a powerful set of graphics drawing calls, and a simple *unalterable* method of delivering events to the owners of the windows. NeXT (if you ignored NextStep) provided this, but they chose to hide it and tried to push their own baroque toolkit instead.

Many of the ideas in FLTK were developed on a NeXT (but *not* using NextStep) in 1987 in a C toolkit Bill called ”views”. Here he came up with passing events downward in the tree and having the handle routine return a value indicating whether it used the event, and the table-driven menus. In general he was trying to prove that complex UI ideas could be entirely implemented in a user space toolkit, with no knowledge or support by the system.

After going to film school for a few years, Bill worked at Sun Microsystems on the (doomed) NeW-S project. Here he found an even better and cleaner windowing system, and he reimplemented ”views” atop that. NeWS did have an unnecessarily complex method of delivering events which hurt it. But the designers did admit that perhaps the user could write just as good of a button as they could, and officially exposed the lower level interface.

With the death of NeWS Bill realized that he would have to live with X. The biggest problem with X is the ”window manager”, which means that the toolkit can no longer control the window borders or drag the window around.

At Digital Domain Bill discovered another toolkit, ”Forms”. Forms was similar to his work, but provided many more widgets, since it was used in many real applications, rather than as theoretical work. He decided to use Forms, except he integrated his table-driven menus into it. Several very large programs were created using this version of Forms.

The need to switch to OpenGL and GLX, portability, and a desire to use C++ subclassing required a rewrite of Forms. This produced the first version of FLTK. The conversion to C++ required so many changes it made it impossible to recompile any Forms objects. Since it was incompatible anyway, Bill decided to incorporate his older ideas as much as possible by simplifying the lower level interface and the event passing mechanism.

Bill received permission to release it for free on the Internet, with the GNU general public license. Response from Internet users indicated that the Linux market dwarfed the SGI and high-speed GL market, so he rewrote it to use X for all drawing, greatly speeding it up on these machines. That is the version you have now.

Digital Domain has since withdrawn support for FLTK. While Bill is no longer able to actively develop it, he still contributes to FLTK in his free time and is a part of the FLTK development team.

3.2 Features

FLTK was designed to be statically linked. This was done by splitting it into many small objects and designing it so that functions that are not used do not have pointers to them in the parts that are used, and thus do not get linked in. This allows you to make an easy-to-install program or to modify FLTK to the exact requirements of your application without worrying about bloat. FLTK works fine as a shared library, though, and is now included with several Linux distributions.

Here are some of the core features unique to FLTK:

- sizeof(Fl_Widget) == 64 to 92.
- The ”core” (the ”hello” program compiled & linked with a static FLTK library using gcc on a 486 and then stripped) is 114K.
- The FLUID program (which includes every widget) is 538k.
- Written directly atop core libraries (Xlib, WIN32 or Cocoa) for maximum speed, and carefully optimized for code size and performance.
- Precise low-level compatibility between the X11, WIN32 and MacOS versions - only about 10% of the code is different.
- Interactive user interface builder program. Output is human-readable and editable C++ source code.
- Support for overlay hardware, with emulation if none is available.
- Very small & fast portable 2-D drawing library to hide Xlib, WIN32, or QuickDraw.
- OpenGL/Mesa drawing area widget.
- Support for OpenGL overlay hardware on both X11 and WIN32, with emulation if none is available.
- Text widgets with cut & paste, undo, and support for Unicode text and international input methods.
- Compatibility header file for the GLUT library.
- Compatibility header file for the XForms library.

3.3 Licensing

FLTK comes with complete free source code. FLTK is available under the terms of the GNU Library General Public License with exceptions that allow for static linking. Contrary to popular belief, it can be used in commercial software - even Bill Gates could use it!

3.4 What Does ”FLTK” Mean?

FLTK was originally designed to be compatible with the Forms Library written for SGI machines. In that library all the functions and structures started with ”fl_”. This naming was extended to all new methods and widgets in the C++ library, and this prefix was taken as the name of the library. It is almost impossible to search for ”FL” on the Internet, due to the fact that it is also the abbreviation for Florida. After much debating and searching for a new name for the toolkit, which was already in use by several people, Bill came up with ”FLTK”, including a bogus excuse that it stands for ”The Fast Light Toolkit”.

3.5 Building and Installing FLTK Under UNIX and Apple OS X

In most cases you can just type "make". This will run configure with the default of no options and then compile everything.

For OS X, Xcode 3 project files can be found in the 'ide' directory.

FLTK uses GNU autoconf to configure itself for your UNIX platform. The main things that the configure script will look for are the X11 and OpenGL (or Mesa) header and library files. If these cannot be found in the standard include/library locations you'll need to define the `CFLAGS`, `CXXFLAGS`, and `LDFLAGS` environment variables. For the Bourne and Korn shells you'd use:

```
CFLAGS=-Iincludedir; export CFLAGS
CXXFLAGS=-Iincludedir; export CXXFLAGS
LDFLAGS=-Llibdir; export LDFLAGS
```

For C shell and tcsh, use:

```
setenv CFLAGS "-Iincludedir"
setenv CXXFLAGS "-Iincludedir"
setenv LDFLAGS "-Llibdir"
```

By default configure will look for a C++ compiler named `CC`, `c++`, `g++`, or `gcc` in that order. To use another compiler you need to set the `CXX` environment variable:

```
CXX=xlC; export CXX
setenv CXX "xlC"
```

The `CC` environment variable can also be used to override the default C compiler (`cc` or `gcc`), which is used for a few FLTK source files.

You can run configure yourself to get the exact setup you need. Type "./configure <options>", where options are:

–enable-cygwin

Enable the Cygwin libraries under WIN32

–enable-debug

Enable debugging code & symbols

–disable-gl

Disable OpenGL support

–enable-shared

Enable generation of shared libraries

–enable-threads

Enable multithreading support

–enable-xdbe

Enable the X double-buffer extension

–enable-xft

Enable the Xft library for anti-aliased fonts under X11

–enable-x11

When targeting cygwin, build with X11 GUI instead of windows GDI

–enable-cp936

Under X11, enable use of the GB2312 locale

–bindir=/path

Set the location for executables [default = $prefix/bin]

–datadir=/path

Set the location for data files. [default = $prefix/share]

–libdir=/path

Set the location for libraries [default = $prefix/lib]

–includedir=/path

Set the location for include files. [default = $prefix/include]

–mandir=/path

Set the location for man pages. [default = $prefix/man]

–prefix=/dir

Set the directory prefix for files [default = /usr/local]

When the configure script is done you can just run the ”make” command. This will build the library, FLUID tool, and all of the test programs.

To install the library, become root and type ”make install”. This will copy the ”fluid” executable to ”bindir”, the header files to ”includedir”, and the library files to ”libdir”.

3.6 Building FLTK Under Microsoft Windows

NOTE: This documentation section is currently under review. More up-to-date information for this release may be available in the file ”README.MSWindows.txt” and you should read that file to determine if there are changes that may be applicable to your build environment.

FLTK 1.3 is officially supported on Windows (2000,) 2003, XP, and later. Older Windows versions prior to Windows 2000 are not officially supported, but may still work. The main reason is that the O-S version needs to support UTF-8. FLTK 1.3 is known to work on recent versions of Windows such as Windows 7, Windows 8/8.1 and Windows 10 and has been reported to work in both 32-bit and 64-bit versions of these.

FLTK currently supports the following development environments on the Windows platform:

CAUTION: Libraries built by any one of these build environments can not be mixed with object files from any of the other environments! (They use incompatible C++ conventions internally.)

Free Microsoft Visual C++ 2008 Express and Visual C++ 2010 Express or later versions using the supplied workspace and project files. Older versions, and the commercial versions, can be used as well, if they can open the project files. Be sure to get your service packs!

The project files can be found in the ”ide/” directory. Please read ”ide/README.IDE” for more info about this.

3.6.1 GNU toolsets (Cygwin or MinGW) hosted on Windows

If using Cygwin with the Cygwin shell, or MinGW with the Msys shell, these build environments behave very much like a Unix or OS X build and the notes above in the section on *Building and Installing FLTK Under UNIX and Apple OS X* apply, in particular the descriptions of using the ”configure” script and its related options.

In general for a build using these tools, e.g. for the Msys shell with MinGW, it should suffice to ”cd” into the directory where you have extracted the fltk tarball and type:

```
./configure
make
```

This will build the fltk libraries and they can then be utilised directly from the build location. NOTE: this may be simpler than ”installing” them in many cases as different tool chains on Windows have different ideas about where the files should be ”installed” to.

For example, if you ”install” the libraries using Msys/MinGW with the following command:

```
make install
```

Then Msys will ”install” the libraries to where it thinks the path ”/usr/local/” leads to. If you only ever build code from within the Msys environment this works well, but the actual ”Windows path” these files are located in will be something like ”C:\msys\1.0\local\lib”, depending on where your Msys installation is rooted, which may not be useful to other tools.

If you want to install your built fltk libraries in a non-standard location you may do:

```
sh configure --prefix=C:/FLTK
make
```

Where the value passed to ”prefix” is the path at which you would like fltk to be installed.

A subsequent invocation of ”make install” will then place the fltk libraries and header files into that path.

The other options to ”configure” may also be used to tailor the build to suit your environment.

3.6.2 Using the Visual C++ DLL Library

The ”fltkdll.dsp” project file builds a DLL-version of the FLTK library. Because of name mangling differences between PC compilers (even between different versions of Visual C++!) you can only use the DLL that is generated with the same version compiler that you built it with.

When compiling an application or DLL that uses the FLTK DLL, you will need to define the `FL_DLL` preprocessor symbol to get the correct linkage commands embedded within the FLTK header files.

3.7 Internet Resources

FLTK is available on the ’net in a bunch of locations:

WWW

http://www.fltk.org/
http://www.fltk.org/str.php [for reporting bugs]
http://www.fltk.org/software.php [source code]
http://www.fltk.org/newsgroups.php [newsgroup/forums]

NNTP Newsgroups

https://groups.google.com/forum/#!forum/fltkgeneral [Google Groups interface]
news://fltk.org:1024/ [NNTP interface]
http://fltk.org/newsgroups.php [web interface]

3.8 Reporting Bugs

To report a bug in FLTK, or for feature requests, please use the form at http://www.fltk.org/str.-php, and click on ”Submit Bug or Feature Request”.

You’ll be prompted for the FLTK version, operating system & version, and compiler that you are using. We will be unable to provide any kind of help without that basic information.

For general support and questions, please use the fltk.general newsgroup (see above, ”NNTP Newsgroups”) or the web interface to the newsgroups at http://fltk.org/newsgroups.php.

Chapter 4

FLTK Basics

This chapter teaches you the basics of compiling programs that use FLTK.

4.1 Writing Your First FLTK Program

All programs must include the file <FL/Fl.H>. In addition the program must include a header file for each FLTK class it uses. Listing 1 shows a simple "Hello, World!" program that uses FLTK to display the window.

Listing 1 - "hello.cxx"

```
#include <FL/Fl.H>
#include <FL/Fl_Window.H>
#include <FL/Fl_Box.H>

int main(int argc, char **argv) {
  Fl_Window *window = new Fl_Window(340,180);
  Fl_Box *box = new Fl_Box(20,40,300,100,"Hello, World!");
  box->box(FL_UP_BOX);
  box->labelfont(FL_BOLD+FL_ITALIC);
  box->labelsize(36);
  box->labeltype(FL_SHADOW_LABEL);
  window->end();
  window->show(argc, argv);
  return Fl::run();
}
```

After including the required header files, the program then creates a window. All following widgets will automatically be children of this window.

```
Fl_Window *window = new Fl_Window(340,180);
```

Then we create a box with the "Hello, World!" string in it. FLTK automatically adds the new box to `window`, the current grouping widget.

```
Fl_Box *box = new Fl_Box(20,40,300,100,"Hello, World!");
```

Next, we set the type of box and the font, size, and style of the label:

```
box->box(FL_UP_BOX);
box->labelfont(FL_BOLD+FL_ITALIC);
box->labelsize(36);
box->labeltype(FL_SHADOW_LABEL);
```

We tell FLTK that we will not add any more widgets to `window`.

```
window->end();
```

Finally, we show the window and enter the FLTK event loop:

```
window->show(argc, argv);
return Fl::run();
```

The resulting program will display the window in Figure 4.1. You can quit the program by closing the window or pressing the ESCape key.

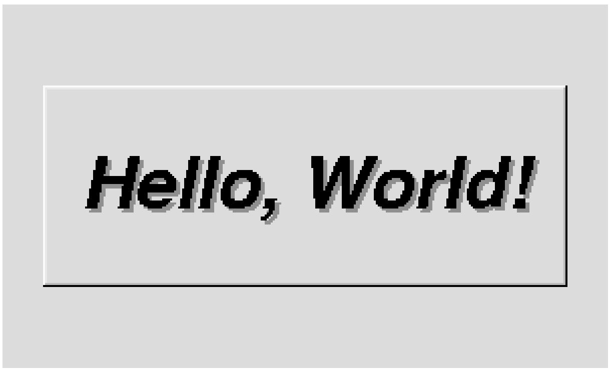


Figure 4.1: The Hello, World! Window

4.1.1 Creating the Widgets

The widgets are created using the C++ `new` operator. For most widgets the arguments to the constructor are:

```
Fl_Widget(x, y, width, height, label)
```

The `x` and `y` parameters determine where the widget or window is placed on the screen. In FLTK the top left corner of the window or screen is the origin (i.e. `x = 0, y = 0`) and the units are in pixels.

The `width` and `height` parameters determine the size of the widget or window in pixels. The maximum widget size is typically governed by the underlying window system or hardware.

`label` is a pointer to a character string to label the widget with or `NULL`. If not specified the label defaults to `NULL`. The label string must be in static storage such as a string constant because FLTK does not make a copy of it - it just uses the pointer.

4.1.2 Creating Widget hierarchies

Widgets are commonly ordered into functional groups, which in turn may be grouped again, creating a hierarchy of widgets. FLTK makes it easy to fill groups by automatically adding all widgets that are created between a `myGroup->begin()` and `myGroup->end()`. In this example, `myGroup` would be the *current* group.

Newly created groups and their derived widgets implicitly call `begin()` in the constructor, effectively adding all subsequently created widgets to itself until `end()` is called.

Setting the current group to `NULL` will stop automatic hierarchies. New widgets can now be added manually using `Fl_Group::add(...)` and `Fl_Group::insert(...)`.

4.1.3 Get/Set Methods

`box->box(FL_UP_BOX)` sets the type of box the Fl_Box draws, changing it from the default of `FL_NO_BOX`, which means that no box is drawn. In our "Hello, World!" example we use `FL_UP_BOX`, which means that a raised button border will be drawn around the widget. More details are available in the Box Types section.

You could examine the boxtype in by doing `box->box()`. FLTK uses method name overloading to make short names for get/set methods. A "set" method is always of the form "void name(type)", and a "get" method is always of the form "type name() const".

4.1.4 Redrawing After Changing Attributes

Almost all of the set/get pairs are very fast, short inline functions and thus very efficient. However, *the "set" methods do not call* `redraw()` - you have to call it yourself. This greatly reduces code size and execution time. The only common exceptions are `value()` which calls `redraw()` and `label()` which calls `redraw_label()` if necessary.

4.1.5 Labels

All widgets support labels. In the case of window widgets, the label is used for the label in the title bar. Our example program calls the `labelfont()`, `labelsize()`, and `labeltype()` methods.

The `labelfont()` method sets the typeface and style that is used for the label, which for this example we are using `FL_BOLD` and `FL_ITALIC`. You can also specify typefaces directly.

The `labelsize()` method sets the height of the font in pixels.

The `labeltype()` method sets the type of label. FLTK supports normal, embossed, and shadowed labels internally, and more types can be added as desired.

A complete list of all label options can be found in the section on Labels and Label Types.

4.1.6 Showing the Window

The `show()` method shows the widget or window. For windows you can also provide the command-line arguments to allow users to customize the appearance, size, and position of your windows.

4.1.7 The Main Event Loop

All FLTK applications (and most GUI applications in general) are based on a simple event processing model. User actions such as mouse movement, button clicks, and keyboard activity generate events that are sent to an application. The application may then ignore the events or respond to the user, typically by redrawing a button in the "down" position, adding the text to an input field, and so forth.

FLTK also supports idle, timer, and file pseudo-events that cause a function to be called when they occur. Idle functions are called when no user input is present and no timers or files need to be handled - in short, when the application is not doing anything. Idle callbacks are often used to update a 3D display or do other background processing.

Timer functions are called after a specific amount of time has expired. They can be used to pop up a progress dialog after a certain amount of time or do other things that need to happen at more-or-less regular intervals. FLTK timers are not 100% accurate, so they should not be used to measure time intervals, for example.

File functions are called when data is ready to read or write, or when an error condition occurs on a file. They are most often used to monitor network connections (sockets) for data-driven displays.

FLTK applications must periodically check (Fl::check()) or wait (Fl::wait()) for events or use the Fl::run() method to enter a standard event processing loop. Calling Fl::run() is equivalent to the following code:

```
while (Fl::wait());
```

Fl::run() does not return until all of the windows under FLTK control are closed by the user or your program.

4.2 Compiling Programs with Standard Compilers

Under UNIX (and under Microsoft Windows when using the GNU development tools) you will probably need to tell the compiler where to find the header files. This is usually done using the `-I` option:

```
CC -I/usr/local/include ...
gcc -I/usr/local/include ...
```

The `fltk-config` script included with FLTK can be used to get the options that are required by your compiler:

```
CC `fltk-config --cxxflags` ...
```

Similarly, when linking your application you will need to tell the compiler to use the FLTK library:

```
CC  ... -L/usr/local/lib -lfltk -lXext -lX11 -lm
gcc ... -L/usr/local/lib -lfltk -lXext -lX11 -lm
```

Aside from the "fltk" library, there is also a "fltk_forms" library for the XForms compatibility classes, "fltk_gl" for the OpenGL and GLUT classes, and "fltk_images" for the image file classes, Fl_Help_Dialog widget, and system icon support.

Note

> The libraries are named "fltk.lib", "fltkgl.lib", "fltkforms.lib", and "fltkimages.lib", respectively under Windows.

As before, the `fltk-config` script included with FLTK can be used to get the options that are required by your linker:

```
CC ... `fltk-config --ldflags`
```

The forms, GL, and images libraries are included with the "--use-foo" options, as follows:

```
CC ... `fltk-config --use-forms --ldflags`
CC ... `fltk-config --use-gl --ldflags`
CC ... `fltk-config --use-images --ldflags`
CC ... `fltk-config --use-forms --use-gl --use-images --ldflags`
```

Finally, you can use the `fltk-config` script to compile a single source file as a FLTK program:

```
fltk-config --compile filename.cpp
fltk-config --use-forms --compile filename.cpp
fltk-config --use-gl --compile filename.cpp
fltk-config --use-images --compile filename.cpp
fltk-config --use-forms --use-gl --use-images --compile filename.cpp
```

Any of these will create an executable named `filename`.

4.3 Compiling Programs with Makefiles

The previous section described how to use `fltk-config` to build a program consisting of a single source file from the command line, and this is very convenient for small test programs. But `fltk-config` can also be used to set the compiler and linker options as variables within a `Makefile` that can be used to build programs out of multiple source files:

```
CXX      = $(shell fltk-config --cxx)
DEBUG    = -g
CXXFLAGS = $(shell fltk-config --use-gl --use-images --cxxflags ) -I.
LDFLAGS  = $(shell fltk-config --use-gl --use-images --ldflags )
LDSTATIC = $(shell fltk-config --use-gl --use-images --ldstaticflags )
LINK     = $(CXX)

TARGET = cube
OBJS = CubeMain.o CubeView.o CubeViewUI.o
SRCS = CubeMain.cxx CubeView.cxx CubeViewUI.cxx

.SUFFIXES: .o .cxx
%.o: %.cxx
        $(CXX) $(CXXFLAGS) $(DEBUG) -c $<

all: $(TARGET)
        $(LINK) -o $(TARGET) $(OBJS) $(LDSTATIC)

$(TARGET): $(OBJS)
CubeMain.o: CubeMain.cxx CubeViewUI.h
CubeView.o: CubeView.cxx CubeView.h CubeViewUI.h
CubeViewUI.o: CubeViewUI.cxx CubeView.h

clean: $(TARGET) $(OBJS)
        rm -f *.o 2> /dev/null
        rm -f $(TARGET) 2> /dev/null
```

4.4 Compiling Programs with Microsoft Visual C++

In Visual C++ you will need to tell the compiler where to find the FLTK header files. This can be done by selecting "Settings" from the "Project" menu and then changing the "Preprocessor" settings under the "C/-C++" tab. You will also need to add the FLTK (`FLTK.LIB` or `FLTKD.LIB`) and the Windows Common Controls (`COMCTL32.LIB`) libraries to the "Link" settings. You must also define `WIN32`.

More information can be found in `README.MSWindows.txt`.

You can build your Microsoft Windows applications as Console or Desktop applications. If you want to use the standard C `main()` function as the entry point, FLTK includes a `WinMain()` function that will call your `main()` function for you.

4.5 Naming

All public symbols in FLTK start with the characters 'F' and 'L':

- Functions are either `Fl::foo()` or `fl_foo()`.
- Class and type names are capitalized: `Fl_Foo`.
- Constants and enumerations are uppercase: `FL_FOO`.
- All header files start with `<FL/...>`.

4.6 Header Files

The proper way to include FLTK header files is:

```
#include <FL/Fl_xyz.H>
```

Note

Case *is* significant on many operating systems, and the C standard uses the forward slash (/) to separate directories. *Do not use any of the following include lines:*

```
#include <FL\Fl_xyz.H>
#include <fl/fl_xyz.h>
#include <Fl/fl_xyz.h>
```

Chapter 5

Common Widgets and Attributes

This chapter describes many of the widgets that are provided with FLTK and covers how to query and set the standard attributes.

5.1 Buttons

FLTK provides many types of buttons:

- Fl_Button - A standard push button.
- Fl_Check_Button - A button with a check box.
- Fl_Light_Button - A push button with a light.
- Fl_Repeat_Button - A push button that repeats when held.
- Fl_Return_Button - A push button that is activated by the `Enter` key.
- Fl_Round_Button - A button with a radio circle.

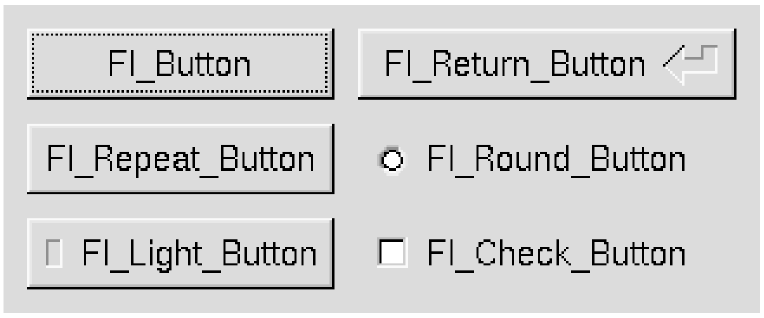


Figure 5.1: FLTK Button Widgets

All of these buttons just need the corresponding `<FL/Fl_xyz_Button.H>` header file. The constructor takes the bounding box of the button and optionally a label string:

```
Fl_Button *button = new Fl_Button(x, y, width, height, "label");
Fl_Light_Button *lbutton = new Fl_Light_Button(x, y, width, height);
Fl_Round_Button *rbutton = new Fl_Round_Button(x, y, width, height, "label");
```

Each button has an associated `type()` which allows it to behave as a push button, toggle button, or radio button:

```
button->type(FL_NORMAL_BUTTON);
lbutton->type(FL_TOGGLE_BUTTON);
rbutton->type(FL_RADIO_BUTTON);
```

For toggle and radio buttons, the `value()` method returns the current button state (0 = off, 1 = on). The `set()` and `clear()` methods can be used on toggle buttons to turn a toggle button on or off, respectively. Radio buttons can be turned on with the `setonly()` method; this will also turn off other radio buttons in the same group.

5.2 Text

FLTK provides several text widgets for displaying and receiving text:

- Fl_Input - A one-line text input field.
- Fl_Output - A one-line text output field.
- Fl_Multiline_Input - A multi-line text input field.
- Fl_Multiline_Output - A multi-line text output field.
- Fl_Text_Display - A multi-line text display widget.
- Fl_Text_Editor - A multi-line text editing widget.
- Fl_Help_View - A HTML text display widget.

The Fl_Output and Fl_Multiline_Output widgets allow the user to copy text from the output field but not change it.

The `value()` method is used to get or set the string that is displayed:

```
Fl_Input *input = new Fl_Input(x, y, width, height, "label");
input->value("Now is the time for all good men...");
```

The string is copied to the widget's own storage when you set the `value()` of the widget.

The Fl_Text_Display and Fl_Text_Editor widgets use an associated Fl_Text_Buffer class for the value, instead of a simple string.

5.3 Valuators

Unlike text widgets, valuators keep track of numbers instead of strings. FLTK provides the following valuators:

- Fl_Counter - A widget with arrow buttons that shows the current value.
- Fl_Dial - A round knob.
- Fl_Roller - An SGI-like dolly widget.
- Fl_Scrollbar - A standard scrollbar widget.
- Fl_Slider - A scrollbar with a knob.
- Fl_Value_Slider - A slider that shows the current value.

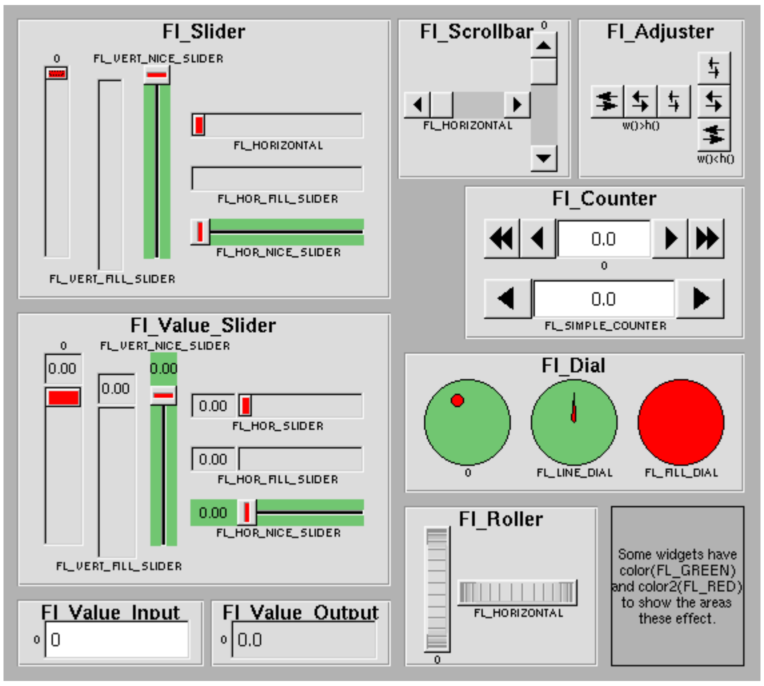


Figure 5.2: FLTK valuator widgets

The `value()` method gets and sets the current value of the widget. The `minimum()` and `maximum()` methods set the range of values that are reported by the widget.

5.4 Groups

The Fl_Group widget class is used as a general purpose ”container” widget. Besides grouping radio buttons, the groups are used to encapsulate windows, tabs, and scrolled windows. The following group classes are available with FLTK:

- Fl_Double_Window - A double-buffered window on the screen.
- Fl_Gl_Window - An OpenGL window on the screen.
- Fl_Group - The base container class; can be used to group any widgets together.
- Fl_Pack - A collection of widgets that are packed into the group area.
- Fl_Scroll - A scrolled window area.
- Fl_Tabs - Displays child widgets as tabs.
- Fl_Tile - A tiled window area.
- Fl_Window - A window on the screen.
- Fl_Wizard - Displays one group of widgets at a time.

5.5 Setting the Size and Position of Widgets

The size and position of widgets is usually set when you create them. You can access them with the `x()`, `y()`, `w()`, and `h()` methods.

You can change the size and position by using the `position()`, `resize()`, and `size()` methods:

```
button->position(x, y);
group->resize(x, y, width, height);
window->size(width, height);
```

If you change a widget's size or position after it is displayed you will have to call `redraw()` on the widget's parent.

5.6 Colors

FLTK stores the colors of widgets as an 32-bit unsigned number that is either an index into a color palette of 256 colors or a 24-bit RGB color. The color palette is *not* the X or MS Windows colormap, but instead is an internal table with fixed contents.

See the Colors section of Drawing Things in FLTK for implementation details.

There are symbols for naming some of the more common colors:

- `FL_BLACK`
- `FL_RED`
- `FL_GREEN`
- `FL_YELLOW`
- `FL_BLUE`
- `FL_MAGENTA`
- `FL_CYAN`
- `FL_WHITE`
- `FL_WHITE`

Other symbols are used as the default colors for all FLTK widgets.

- `FL_FOREGROUND_COLOR`
- `FL_BACKGROUND_COLOR`
- `FL_INACTIVE_COLOR`
- `FL_SELECTION_COLOR`

The full list of named color values can be found in FLTK Enumerations.

A color value can be created from its RGB components by using the `fl_rgb_color()` function, and decomposed again with `Fl::get_color()`:

```
Fl_Color c = fl_rgb_color(85, 170, 255);     // RGB to Fl_Color
Fl::get_color(c, r, g, b);                    // Fl_Color to RGB
```

The widget color is set using the `color()` method:

```
button->color(FL_RED);                        // set color using named value
```

Similarly, the label color is set using the `labelcolor()` method:

```
button->labelcolor(FL_WHITE);
```

The Fl_Color encoding maps to a 32-bit unsigned integer representing RGBI, so it is also possible to specify a color using a hex constant as a color map index:

```
button->color(0x000000ff);                  // colormap index #255 (FL_WHITE)
```

or specify a color using a hex constant for the RGB components:

```
button->color(0xff000000);                  // RGB: red
button->color(0x00ff0000);                  // RGB: green
button->color(0x0000ff00);                  // RGB: blue
button->color(0xffffff00);                  // RGB: white
```

Note

If TrueColor is not available, any RGB colors will be set to the nearest entry in the colormap.

5.7 Box Types

The type Fl_Boxtype stored and returned in Fl_Widget::box() is an enumeration defined in Enumerations.H.

Figure 3-3 shows the standard box types included with FLTK.

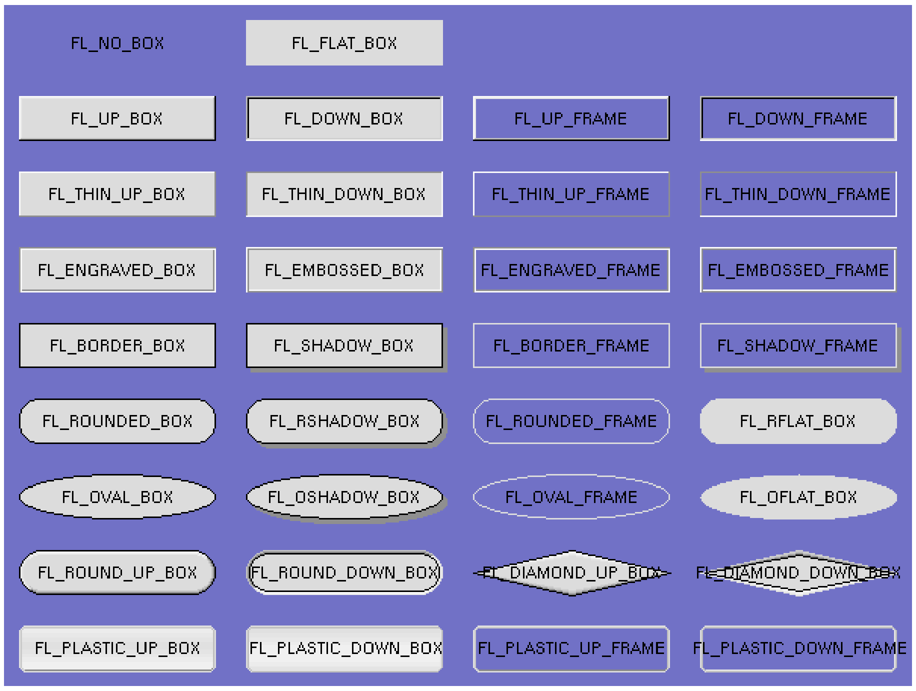


Figure 5.3: FLTK box types

FL_NO_BOX means nothing is drawn at all, so whatever is already on the screen remains. The FL_-..._FRAME types only draw their edges, leaving the interior unchanged. The blue color in Figure 3-3 is the area that is not drawn by the frame types.

5.7.1 Making Your Own Boxtypes

You can define your own boxtypes by making a small function that draws the box and adding it to the table of boxtypes.

The Drawing Function

The drawing function is passed the bounding box and background color for the widget:

```
void xyz_draw(int x, int y, int w, int h, Fl_Color c) {
...
}
```

A simple drawing function might fill a rectangle with the given color and then draw a black outline:

```
void xyz_draw(int x, int y, int w, int h, Fl_Color c) {
  fl_color(c);
  fl_rectf(x, y, w, h);
  fl_color(FL_BLACK);
  fl_rect(x, y, w, h);
}
```

Fl_Boxtype fl_down(Fl_Boxtype b)

fl_down() returns the "pressed" or "down" version of a box. If no "down" version of a given box exists, the behavior of this function is undefined and some random box or frame is returned. See Drawing Functions for more details.

Fl_Boxtype fl_frame(Fl_Boxtype b)

fl_frame() returns the unfilled, frame-only version of a box. If no frame version of a given box exists, the behavior of this function is undefined and some random box or frame is returned. See Drawing Functions for more details.

Fl_Boxtype fl_box(Fl_Boxtype b)

fl_box() returns the filled version of a frame. If no filled version of a given frame exists, the behavior of this function is undefined and some random box or frame is returned. See Drawing Functions for more details.

Adding Your Box Type

The Fl::set_boxtype() method adds or replaces the specified box type:

```
#define XYZ_BOX FL_FREE_BOXTYPE

Fl::set_boxtype(XYZ_BOX, xyz_draw, 1, 1, 2, 2);
```

The last 4 arguments to Fl::set_boxtype() are the offsets for the `x`, `y`, `width`, and `height` values that should be subtracted when drawing the label inside the box.

A complete box design contains four box types in this order: a filled, neutral box (`UP_BOX`), a filled, depressed box (`DOWN_BOX`), and the same as outlines only (`UP_FRAME` and `DOWN_FRAME`). The function fl_down(Fl_Boxtype) expects the neutral design on a boxtype with a numerical value evenly dividable by two. fl_frame(Fl_Boxtype) expects the `UP_BOX` design at a value dividable by four.

5.8 Labels and Label Types

The `label()`, `align()`, `labelfont()`, `labelsize()`, `labeltype()`, `image()`, and `deimage()` methods control the labeling of widgets.

label()

The `label()` method sets the string that is displayed for the label. Symbols can be included with the label string by escaping them using the "@" symbol - "@@" displays a single at sign. Figure 3-4 shows the available symbols.

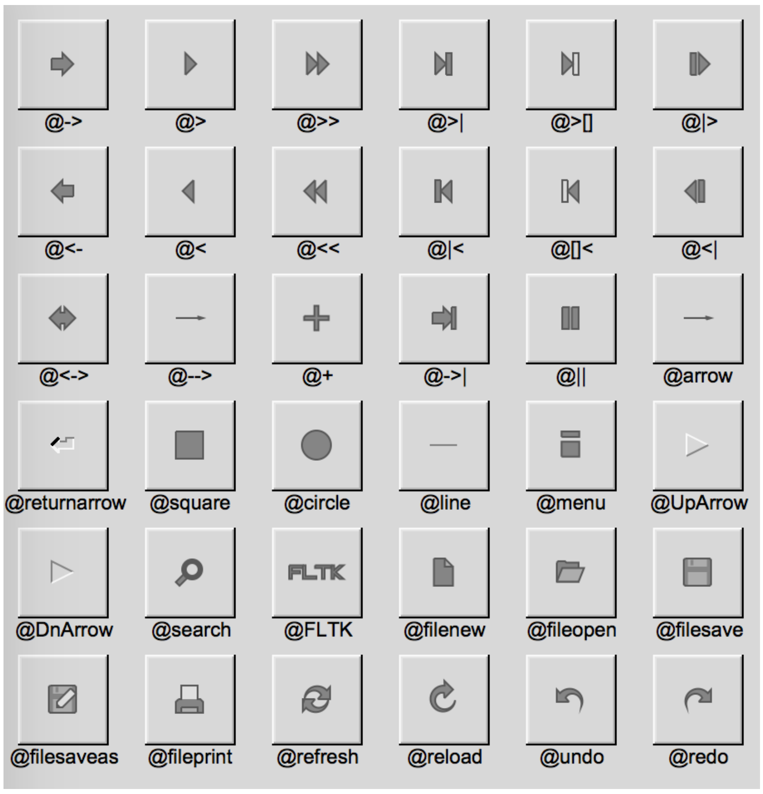


Figure 5.4: FLTK label symbols

The @ sign may also be followed by the following optional "formatting" characters, in this order:

- '#' forces square scaling, rather than distortion to the widget's shape.
- +[1-9] or -[1-9] tweaks the scaling a little bigger or smaller.
- '$' flips the symbol horizontally, '%' flips it vertically.
- [0-9] - rotates by a multiple of 45 degrees. '5' and '6' do no rotation while the others point in the direction of that key on a numeric keypad. '0', followed by four more digits rotates the symbol by that amount in degrees.

Thus, to show a very large arrow pointing downward you would use the label string "@+92->".

align()

The `align()` method positions the label. The following constants are defined and may be OR'd together as needed:

- `FL_ALIGN_CENTER` - center the label in the widget.
- `FL_ALIGN_TOP` - align the label at the top of the widget.
- `FL_ALIGN_BOTTOM` - align the label at the bottom of the widget.
- `FL_ALIGN_LEFT` - align the label to the left of the widget.
- `FL_ALIGN_RIGHT` - align the label to the right of the widget.
- `FL_ALIGN_LEFT_TOP` - The label appears to the left of the widget, aligned at the top. Outside labels only.
- `FL_ALIGN_RIGHT_TOP` - The label appears to the right of the widget, aligned at the top. Outside labels only.
- `FL_ALIGN_LEFT_BOTTOM` - The label appears to the left of the widget, aligned at the bottom. Outside labels only.
- `FL_ALIGN_RIGHT_BOTTOM` - The label appears to the right of the widget, aligned at the bottom. Outside labels only.
- `FL_ALIGN_INSIDE` - align the label inside the widget.
- `FL_ALIGN_CLIP` - clip the label to the widget's bounding box.
- `FL_ALIGN_WRAP` - wrap the label text as needed.
- `FL_ALIGN_TEXT_OVER_IMAGE` - show the label text over the image.
- `FL_ALIGN_IMAGE_OVER_TEXT` - show the label image over the text (default).
- `FL_ALIGN_IMAGE_NEXT_TO_TEXT` - The image will appear to the left of the text.
- `FL_ALIGN_TEXT_NEXT_TO_IMAGE` - The image will appear to the right of the text.
- `FL_ALIGN_IMAGE_BACKDROP` - The image will be used as a background for the widget.

labeltype()

The `labeltype()` method sets the type of the label. The following standard label types are included:

- `FL_NORMAL_LABEL` - draws the text.
- `FL_NO_LABEL` - does nothing.
- `FL_SHADOW_LABEL` - draws a drop shadow under the text.
- `FL_ENGRAVED_LABEL` - draws edges as though the text is engraved.
- `FL_EMBOSSED_LABEL` - draws edges as thought the text is raised.
- `FL_ICON_LABEL` - draws the icon associated with the text.

image() and deimage()

The image() and deimage() methods set an image that will be displayed with the widget. The deimage() method sets the image that is shown when the widget is inactive, while the image() method sets the image that is shown when the widget is active.

To make an image you use a subclass of Fl_Image.

Making Your Own Label Types

Label types are actually indexes into a table of functions that draw them. The primary purpose of this is to use this to draw the labels in ways inaccessible through the fl_font() mechanism (e.g. FL_ENGRAVED_LABEL) or with program-generated letters or symbology.

Label Type Functions

To setup your own label type you will need to write two functions: one to draw and one to measure the label. The draw function is called with a pointer to a Fl_Label structure containing the label information, the bounding box for the label, and the label alignment:

```
void xyz_draw(const Fl_Label *label, int x, int y, int w, int h,
        Fl_Align align) {
...
}
```

The label should be drawn *inside* this bounding box, even if FL_ALIGN_INSIDE is not enabled. The function is not called if the label value is NULL.

The measure function is called with a pointer to a Fl_Label structure and references to the width and height:

```
void xyz_measure(const Fl_Label *label, int &w, int &h) {
...
}
```

The function should measure the size of the label and set w and h to the size it will occupy.

Adding Your Label Type

The Fl::set_labeltype() method creates a label type using your draw and measure functions:

```
#define XYZ_LABEL FL_FREE_LABELTYPE

Fl::set_labeltype(XYZ_LABEL, xyz_draw, xyz_measure);
```

The label type number n can be any integer value starting at the constant FL_FREE_LABELTYPE. Once you have added the label type you can use the labeltype() method to select your label type.

The Fl::set_labeltype() method can also be used to overload an existing label type such as FL_NORMAL_LABEL.

Making your own symbols

It is also possible to define your own drawings and add them to the symbol list, so they can be rendered as part of any label.

To create a new symbol, you implement a drawing function void drawit(Fl_Color c) which typically uses the functions described in Drawing Complex Shapes to generate a vector shape inside a two-by-two units sized box around the origin. This function is then linked into the symbols table using fl_add_symbol():

```
int fl_add_symbol(const char *name, void (*drawit)(Fl_Color), int scalable)
```

name is the name of the symbol without the "@"; scalable must be set to 1 if the symbol is generated using scalable vector drawing functions.

```
int fl_draw_symbol(const char *name,int x,int y,int w,int h,
      Fl_Color col)
```

This function draws a named symbol fitting the given rectangle.

5.9 Callbacks

Callbacks are functions that are called when the value of a widget changes. A callback function is sent a Fl_Widget pointer of the widget that changed and a pointer to data that you provide:

```
void xyz_callback(Fl_Widget *w, void *data) {
...
}
```

The callback() method sets the callback function for a widget. You can optionally pass a pointer to some data needed for the callback:

```
int xyz_data;

button->callback(xyz_callback, &xyz_data);
```

Normally callbacks are performed only when the value of the widget changes. You can change this using the Fl_Widget::when() method:

```
button->when(FL_WHEN_NEVER);
button->when(FL_WHEN_CHANGED);
button->when(FL_WHEN_RELEASE);
button->when(FL_WHEN_RELEASE_ALWAYS);
button->when(FL_WHEN_ENTER_KEY);
button->when(FL_WHEN_ENTER_KEY_ALWAYS);
button->when(FL_WHEN_CHANGED | FL_WHEN_NOT_CHANGED);
```

Note:
You cannot delete a widget inside a callback, as the widget may still be accessed by FLTK after your callback is completed. Instead, use the Fl::delete_widget() method to mark your widget for deletion when it is safe to do so.
Hint:
Many programmers new to FLTK or C++ try to use a non-static class method instead of a static class method or function for their callback. Since callbacks are done outside a C++ class, the this pointer is not initialized for class methods.
To work around this problem, define a static method in your class that accepts a pointer to the class, and then have the static method call the class method(s) as needed. The data pointer you provide to the callback() method of the widget can be a pointer to the instance of your class.

```
class Foo {
  void my_callback(Fl_Widget *w);
  static void my_static_callback(Fl_Widget *w, void *f) { ((Foo *)f)->my_callback(w); }
  ...
}

...

w->callback(my_static_callback, (void *)this);
```

5.10 Shortcuts

Shortcuts are key sequences that activate widgets such as buttons or menu items. The `shortcut()` method sets the shortcut for a widget:

```
button->shortcut(FL_Enter);
button->shortcut(FL_SHIFT + 'b');
button->shortcut(FL_CTRL + 'b');
button->shortcut(FL_ALT + 'b');
button->shortcut(FL_CTRL + FL_ALT + 'b');
button->shortcut(0); // no shortcut
```

The shortcut value is the key event value - the ASCII value or one of the special keys described in Fl::event_key() Values combined with any modifiers like `Shift` , `Alt` , and `Control`.

Chapter 6

Designing a Simple Text Editor

This chapter takes you through the design of a simple FLTK-based text editor.

6.1 Determining the Goals of the Text Editor

Since this will be the first big project you'll be doing with FLTK, lets define what we want our text editor to do:

1. Provide a menubar/menus for all functions.
2. Edit a single text file, possibly with multiple views.
3. Load from a file.
4. Save to a file.
5. Cut/copy/delete/paste functions.
6. Search and replace functions.
7. Keep track of when the file has been changed.

6.2 Designing the Main Window

Now that we've outlined the goals for our editor, we can begin with the design of our GUI. Obviously the first thing that we need is a window, which we'll place inside a class called `EditorWindow`:

```
class EditorWindow : public Fl_Double_Window {
  public:
    EditorWindow(int w, int h, const char* t);
    ~EditorWindow();

    Fl_Window          *replace_dlg;
    Fl_Input           *replace_find;
    Fl_Input           *replace_with;
    Fl_Button          *replace_all;
    Fl_Return_Button   *replace_next;
    Fl_Button          *replace_cancel;

    Fl_Text_Editor     *editor;
    char               search[256];
};
```

6.3 Variables

Our text editor will need some global variables to keep track of things:

```
int             changed = 0;
char            filename[256] = "";
Fl_Text_Buffer *textbuf;
```

The `textbuf` variable is the text editor buffer for our window class described previously. We'll cover the other variables as we build the application.

6.4 Menubars and Menus

The first goal requires us to use a menubar and menus that define each function the editor needs to perform. The Fl_Menu_Item structure is used to define the menus and items in a menubar:

```
Fl_Menu_Item menuitems[] = {
  { "&File",              0, 0, 0, FL_SUBMENU },
    { "&New File",        0, (Fl_Callback *)new_cb },
    { "&Open File...",    FL_COMMAND + 'o', (Fl_Callback *)open_cb },
    { "&Insert File...",  FL_COMMAND + 'i', (Fl_Callback *)insert_cb, 0,
      FL_MENU_DIVIDER },
    { "&Save File",       FL_COMMAND + 's', (Fl_Callback *)save_cb },
    { "Save File &As...", FL_COMMAND + FL_SHIFT + 's', (
      Fl_Callback *)saveas_cb, 0, FL_MENU_DIVIDER },
    { "New &View", FL_ALT + 'v', (Fl_Callback *)view_cb, 0 },
    { "&Close View", FL_COMMAND + 'w', (Fl_Callback *)close_cb, 0,
      FL_MENU_DIVIDER },
    { "E&xit", FL_COMMAND + 'q', (Fl_Callback *)quit_cb, 0 },
    { 0 },

  { "&Edit", 0, 0, 0, FL_SUBMENU },
    { "&Undo",       FL_COMMAND + 'z', (Fl_Callback *)undo_cb, 0,
      FL_MENU_DIVIDER },
    { "Cu&t",        FL_COMMAND + 'x', (Fl_Callback *)cut_cb },
    { "&Copy",       FL_COMMAND + 'c', (Fl_Callback *)copy_cb },
    { "&Paste",      FL_COMMAND + 'v', (Fl_Callback *)paste_cb },
    { "&Delete",     0, (Fl_Callback *)delete_cb },
    { 0 },

  { "&Search", 0, 0, 0, FL_SUBMENU },
    { "&Find...",       FL_COMMAND + 'f', (Fl_Callback *)find_cb },
    { "F&ind Again",    FL_COMMAND + 'g', find2_cb },
    { "&Replace...",    FL_COMMAND + 'r', replace_cb },
    { "Re&place Again", FL_COMMAND + 't', replace2_cb },
    { 0 },

  { 0 }
};
```

Once we have the menus defined we can create the Fl_Menu_Bar widget and assign the menus to it with:

```
Fl_Menu_Bar *m = new Fl_Menu_Bar(0, 0, 640, 30);
m->copy(menuitems);
```

We'll define the callback functions later.

6.5 Editing the Text

To keep things simple our text editor will use the Fl_Text_Editor widget to edit the text:

```
w->editor = new Fl_Text_Editor(0, 30, 640, 370);
w->editor->buffer(textbuf);
```

So that we can keep track of changes to the file, we also want to add a ”modify” callback:

```
textbuf->add_modify_callback(changed_cb, w);
textbuf->call_modify_callbacks();
```

Finally, we want to use a mono-spaced font like `FL_COURIER`:

```
w->editor->textfont(FL_COURIER);
```

6.6 The Replace Dialog

We can use the FLTK convenience functions for many of the editor's dialogs, however the replace dialog needs its own custom window. To keep things simple we will have a "find" string, a "replace" string, and "replace all", "replace next", and "cancel" buttons. The strings are just Fl_Input widgets, the "replace all" and "cancel" buttons are Fl_Button widgets, and the "replace next " button is a Fl_Return_Button widget:

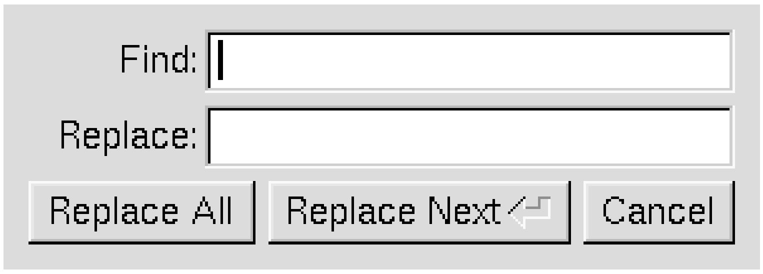


Figure 6.1: The search and replace dialog

```
Fl_Window *replace_dlg = new Fl_Window(300, 105, "Replace");
Fl_Input *replace_find = new Fl_Input(70, 10, 200, 25, "Find:");
Fl_Input *replace_with = new Fl_Input(70, 40, 200, 25, "Replace:");
Fl_Button *replace_all = new Fl_Button(10, 70, 90, 25, "Replace All");
Fl_Button *replace_next = new Fl_Button(105, 70, 120, 25, "Replace Next");
Fl_Button *replace_cancel = new Fl_Button(230, 70, 60, 25, "Cancel");
```

6.7 Callbacks

Now that we've defined the GUI components of our editor, we need to define our callback functions.

6.7.1 changed_cb()

This function will be called whenever the user changes any text in the `editor` widget:

```
void changed_cb(int, int nInserted, int nDeleted,int, const char*, void* v) {
  if ((nInserted || nDeleted) && !loading) changed = 1;
  EditorWindow *w = (EditorWindow *)v;
  set_title(w);
  if (loading) w->editor->show_insert_position();
}
```

The `set_title()` function is one that we will write to set the changed status on the current file. We're doing it this way because we want to show the changed status in the window's title bar.

6.7.2 copy_cb()

This callback function will call Fl_Text_Editor::kf_copy() to copy the currently selected text to the clipboard-:

```
void copy_cb(Fl_Widget*, void* v) {
  EditorWindow* e = (EditorWindow*)v;
  Fl_Text_Editor::kf_copy(0, e->editor);
}
```

6.7.3 cut_cb()

This callback function will call Fl_Text_Editor::kf_cut() to cut the currently selected text to the clipboard:

```
void cut_cb(Fl_Widget*, void* v) {
  EditorWindow* e = (EditorWindow*)v;
  Fl_Text_Editor::kf_cut(0, e->editor);
}
```

6.7.4 delete_cb()

This callback function will call Fl_Text_Buffer::remove_selection() to delete the currently selected text to the clipboard:

```
void delete_cb(Fl_Widget*, void* v) {
  textbuf->remove_selection();
}
```

6.7.5 find_cb()

This callback function asks for a search string using the fl_input() convenience function and then calls the `find2_cb()` function to find the string:

```
void find_cb(Fl_Widget* w, void* v) {
  EditorWindow* e = (EditorWindow*)v;
  const char *val;

  val = fl_input("Search String:", e->search);
  if (val != NULL) {
    // User entered a string - go find it!
    strcpy(e->search, val);
    find2_cb(w, v);
  }
```

6.7.6 find2_cb()

This function will find the next occurrence of the search string. If the search string is blank then we want to pop up the search dialog:

```
void find2_cb(Fl_Widget* w, void* v) {
  EditorWindow* e = (EditorWindow*)v;
  if (e->search[0] == '\0') {
    // Search string is blank; get a new one...
    find_cb(w, v);
    return;
  }

  int pos = e->editor->insert_position();
  int found = textbuf->search_forward(pos, e->search, &pos);
  if (found) {
    // Found a match; select and update the position...
    textbuf->select(pos, pos+strlen(e->search));
    e->editor->insert_position(pos+strlen(e->search));
    e->editor->show_insert_position();
  }
  else fl_alert("No occurrences of \'%s\' found!", e->search);
}
```

If the search string cannot be found we use the fl_alert() convenience function to display a message to that effect.

6.7.7 new_cb()

This callback function will clear the editor widget and current filename. It also calls the `check_save()` function to give the user the opportunity to save the current file first as needed:

```
void new_cb(Fl_Widget*, void*) {
  if (!check_save()) return;

  filename[0] = '\0';
  textbuf->select(0, textbuf->length());
  textbuf->remove_selection();
  changed = 0;
  textbuf->call_modify_callbacks();
}
```

6.7.8 open_cb()

This callback function will ask the user for a filename and then load the specified file into the input widget and current filename. It also calls the `check_save()` function to give the user the opportunity to save the current file first as needed:

```
void open_cb(Fl_Widget*, void*) {
  if (!check_save()) return;

  char *newfile = fl_file_chooser("Open File?", "*", filename);
  if (newfile != NULL) load_file(newfile, -1);
}
```

We call the `load_file()` function to actually load the file.

6.7.9 paste_cb()

This callback function will call Fl_Text_Editor::kf_paste() to paste the clipboard at the current position:

```
void paste_cb(Fl_Widget*, void* v) {
  EditorWindow* e = (EditorWindow*)v;
  Fl_Text_Editor::kf_paste(0, e->editor);
}
```

6.7.10 quit_cb()

The quit callback will first see if the current file has been modified, and if so give the user a chance to save it. It then exits from the program:

```
void quit_cb(Fl_Widget*, void*) {
  if (changed && !check_save())
    return;

  exit(0);
}
```

6.7.11 replace_cb()

The replace callback just shows the replace dialog:

```
void replace_cb(Fl_Widget*, void* v) {
  EditorWindow* e = (EditorWindow*)v;
  e->replace_dlg->show();
}
```

6.7.12 replace2_cb()

This callback will replace the next occurrence of the replacement string. If nothing has been entered for the replacement string, then the replace dialog is displayed instead:

```
void replace2_cb(Fl_Widget*, void* v) {
  EditorWindow* e = (EditorWindow*)v;
  const char *find = e->replace_find->value();
  const char *replace = e->replace_with->value();

  if (find[0] == '\0') {
    // Search string is blank; get a new one...
    e->replace_dlg->show();
    return;
  }

  e->replace_dlg->hide();

  int pos = e->editor->insert_position();
  int found = textbuf->search_forward(pos, find, &pos);

  if (found) {
    // Found a match; update the position and replace text...
    textbuf->select(pos, pos+strlen(find));
    textbuf->remove_selection();
    textbuf->insert(pos, replace);
    textbuf->select(pos, pos+strlen(replace));
    e->editor->insert_position(pos+strlen(replace));
    e->editor->show_insert_position();
  }
  else fl_alert("No occurrences of \'%s\' found!", find);
}
```

6.7.13 replall_cb()

This callback will replace all occurrences of the search string in the file:

```
void replall_cb(Fl_Widget*, void* v) {
  EditorWindow* e = (EditorWindow*)v;
  const char *find = e->replace_find->value();
  const char *replace = e->replace_with->value();

  find = e->replace_find->value();
  if (find[0] == '\0') {
    // Search string is blank; get a new one...
    e->replace_dlg->show();
    return;
  }

  e->replace_dlg->hide();

  e->editor->insert_position(0);
  int times = 0;

  // Loop through the whole string
  for (int found = 1; found;) {
    int pos = e->editor->insert_position();
    found = textbuf->search_forward(pos, find, &pos);

    if (found) {
      // Found a match; update the position and replace text...
      textbuf->select(pos, pos+strlen(find));
      textbuf->remove_selection();
      textbuf->insert(pos, replace);
      e->editor->insert_position(pos+strlen(replace));
      e->editor->show_insert_position();
      times++;
    }
  }

  if (times) fl_message("Replaced %d occurrences.", times);
  else fl_alert("No occurrences of \'%s\' found!", find);
}
```

6.7.14 replcan_cb()

This callback just hides the replace dialog:

```
void replcan_cb(Fl_Widget*, void* v) {
  EditorWindow* e = (EditorWindow*)v;
  e->replace_dlg->hide();
}
```

6.7.15 save_cb()

This callback saves the current file. If the current filename is blank it calls the ”save as” callback:

```
void save_cb(void) {
  if (filename[0] == '\0') {
    // No filename - get one!
    saveas_cb();
    return;
  }
  else save_file(filename);
}
```

The `save_file()` function saves the current file to the specified filename.

6.7.16 saveas_cb()

This callback asks the user for a filename and saves the current file:

```
void saveas_cb(void) {
  char *newfile;

  newfile = fl_file_chooser("Save File As?", "*", filename);
  if (newfile != NULL) save_file(newfile);
}
```

The `save_file()` function saves the current file to the specified filename.

6.8 Other Functions

Now that we've defined the callback functions, we need our support functions to make it all work:

6.8.1 check_save()

This function checks to see if the current file needs to be saved. If so, it asks the user if they want to save it:

```
int check_save(void) {
  if (!changed) return 1;

  int r = fl_choice("The current file has not been saved.\n"
                    "Would you like to save it now?",
                    "Cancel", "Save", "Discard");

  if (r == 1) {
    save_cb(); // Save the file...
    return !changed;
  }

  return (r == 2) ? 1 : 0;
}
```

6.8.2 load_file()

This function loads the specified file into the `textbuf` variable:

```
int loading = 0;
void load_file(char *newfile, int ipos) {
  loading = 1;
  int insert = (ipos != -1);
  changed = insert;
  if (!insert) strcpy(filename, "");
  int r;
  if (!insert) r = textbuf->loadfile(newfile);
  else r = textbuf->insertfile(newfile, ipos);
  if (r)
    fl_alert("Error reading from file \'%s\':\n%s.", newfile, strerror(errno));
  else
    if (!insert) strcpy(filename, newfile);
  loading = 0;
  textbuf->call_modify_callbacks();
}
```

When loading the file we use the Fl_Text_Buffer::loadfile() method to "replace" the text in the buffer, or the Fl_Text_Buffer::insertfile() method to insert text in the buffer from the named file.

6.8.3 save_file()

This function saves the current buffer to the specified file:

```
void save_file(char *newfile) {
  if (textbuf->savefile(newfile))
    fl_alert("Error writing to file \'%s\':\n%s.", newfile, strerror(errno));
  else
    strcpy(filename, newfile);
  changed = 0;
  textbuf->call_modify_callbacks();
}
```

6.8.4 set_title()

This function checks the `changed` variable and updates the window label accordingly:

```
void set_title(Fl_Window* w) {
  if (filename[0] == '\0') strcpy(title, "Untitled");
  else {
    char *slash;
    slash = strrchr(filename, '/');
#ifdef WIN32
    if (slash == NULL) slash = strrchr(filename, '\\');
#endif
    if (slash != NULL) strcpy(title, slash + 1);
    else strcpy(title, filename);
  }

  if (changed) strcat(title, " (modified)");

  w->label(title);
}
```

6.9 The main() Function

Once we've created all of the support functions, the only thing left is to tie them all together with the `main()` function. The `main()` function creates a new text buffer, creates a new view (window) for the text, shows the window, loads the file on the command-line (if any), and then enters the FLTK event loop:

```
int main(int argc, char **argv) {
  textbuf = new Fl_Text_Buffer;

  Fl_Window* window = new_view();

  window->show(1, argv);

  if (argc > 1) load_file(argv[1], -1);

  return Fl::run();
}
```

6.10 Compiling the Editor

The complete source for our text editor can be found in the `test/editor.cxx` source file. Both the Makefile and Visual C++ workspace include the necessary rules to build the editor. You can also compile it using a standard compiler with:

```
CC -o editor editor.cxx -lfltk -lXext -lX11 -lm
```

or by using the `fltk-config` script with:

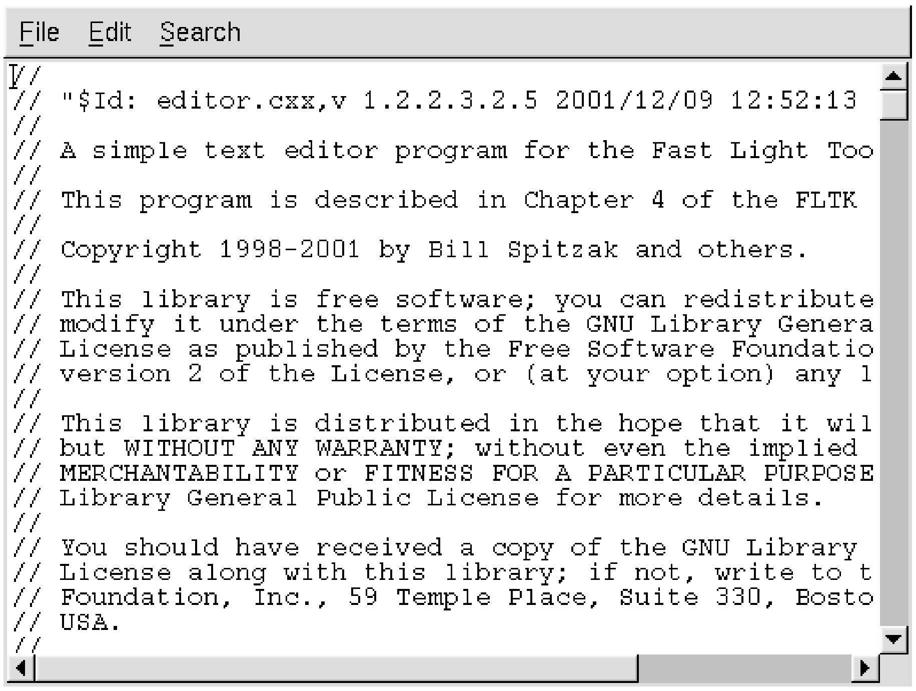
```
fltk-config --compile editor.cxx
```

As noted in Compiling Programs with Standard Compilers, you may need to include compiler and linker options to tell them where to find the FLTK library. Also, the `CC` command may also be called `gcc` or `c++` on your system.

Congratulations, you've just built your own text editor!

6.11 The Final Product

The final editor window should look like the image in Figure 4-2.



```
//
// "$Id: editor.cxx,v 1.2.2.3.2.5 2001/12/09 12:52:13
//
// A simple text editor program for the Fast Light Too
//
// This program is described in Chapter 4 of the FLTK
//
// Copyright 1998-2001 by Bill Spitzak and others.
//
// This library is free software; you can redistribute
// modify it under the terms of the GNU Library Genera
// License as published by the Free Software Foundatio
// version 2 of the License, or (at your option) any l
//
// This library is distributed in the hope that it wil
// but WITHOUT ANY WARRANTY; without even the implied
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE
// Library General Public License for more details.
//
// You should have received a copy of the GNU Library
// License along with this library; if not, write to t
// Foundation, Inc., 59 Temple Place, Suite 330, Bosto
// USA.
//
```

Figure 6.2: The completed editor window

6.12 Advanced Features

Now that we've implemented the basic functionality, it is time to show off some of the advanced features of the Fl_Text_Editor widget.

6.12.1 Syntax Highlighting

The Fl_Text_Editor widget supports highlighting of text with different fonts, colors, and sizes. The implementation is based on the excellent NEdit text editor core, from http://www.nedit.org/, which uses a parallel "style" buffer which tracks the font, color, and size of the text that is drawn.

Styles are defined using the Fl_Text_Display::Style_Table_Entry structure defined in <FL/Fl_Text_Display.H>:

```
struct Style_Table_Entry {
  Fl_Color color;
  Fl_Font  font;
  int      size;
  unsigned attr;
};
```

The color member sets the color for the text, the font member sets the FLTK font index to use, and the size member sets the pixel size of the text. The attr member is currently not used.

For our text editor we'll define 7 styles for plain code, comments, keywords, and preprocessor directives-:

```
Fl_Text_Display::Style_Table_Entry styletable[] = {     // Style table
  { FL_BLACK,      FL_COURIER,        FL_NORMAL_SIZE }, // A - Plain
  { FL_DARK_GREEN, FL_COURIER_ITALIC, FL_NORMAL_SIZE }, // B - Line comments
  { FL_DARK_GREEN, FL_COURIER_ITALIC, FL_NORMAL_SIZE }, // C - Block
      comments
  { FL_BLUE,       FL_COURIER,        FL_NORMAL_SIZE }, // D - Strings
  { FL_DARK_RED,   FL_COURIER,        FL_NORMAL_SIZE }, // E - Directives
  { FL_DARK_RED,   FL_COURIER_BOLD,   FL_NORMAL_SIZE }, // F - Types
  { FL_BLUE,       FL_COURIER_BOLD,   FL_NORMAL_SIZE }  // G - Keywords
};
```

You'll notice that the comments show a letter next to each style - each style in the style buffer is referenced using a character starting with the letter 'A'.

You call the `highlight_data()` method to associate the style data and buffer with the text editor widget:

```
Fl_Text_Buffer *stylebuf;

w->editor->highlight_data(stylebuf, styletable,
                          sizeof(styletable) / sizeof(styletable[0]),
                          'A', style_unfinished_cb, 0);
```

Finally, you need to add a callback to the main text buffer so that changes to the text buffer are mirrored in the style buffer:

```
textbuf->add_modify_callback(style_update, w->editor);
```

The `style_update()` function, like the `change_cb()` function described earlier, is called whenever text is added or removed from the text buffer. It mirrors the changes in the style buffer and then updates the style data as necessary:

```
//
// 'style_update()' - Update the style buffer...
//

void
style_update(int        pos,          // I - Position of update
             int        nInserted,    // I - Number of inserted chars
             int        nDeleted,     // I - Number of deleted chars
             int        nRestyled,    // I - Number of restyled chars
             const char *deletedText, // I - Text that was deleted
             void       *cbArg) {     // I - Callback data
  int  start,                         // Start of text
       end;                           // End of text
  char last,                          // Last style on line
        style,                        // Style data
        text;                         // Text data


  // If this is just a selection change, just unselect the style buffer...
  if (nInserted == 0 && nDeleted == 0) {
    stylebuf->unselect();
    return;
  }

  // Track changes in the text buffer...
  if (nInserted > 0) {
    // Insert characters into the style buffer...
    style = new char[nInserted + 1];
    memset(style, 'A', nInserted);
    style[nInserted] = '\0';

    stylebuf->replace(pos, pos + nDeleted, style);
    delete[] style;
  } else {
    // Just delete characters in the style buffer...
    stylebuf->remove(pos, pos + nDeleted);
  }

  // Select the area that was just updated to avoid unnecessary
  // callbacks...
  stylebuf->select(pos, pos + nInserted - nDeleted);

  // Re-parse the changed region; we do this by parsing from the
  // beginning of the line of the changed region to the end of
  // the line of the changed region...  Then we check the last
  // style character and keep updating if we have a multi-line
  // comment character...
  start = textbuf->line_start(pos);
  end   = textbuf->line_end(pos + nInserted - nDeleted);
  text  = textbuf->text_range(start, end);
  style = stylebuf->text_range(start, end);
  last  = style[end - start - 1];

  style_parse(text, style, end - start);

  stylebuf->replace(start, end, style);
  ((Fl_Text_Editor *)cbArg)->redisplay_range(start, end);

  if (last != style[end - start - 1]) {
    // The last character on the line changed styles, so reparse the
    // remainder of the buffer...
    free(text);
    free(style);

    end   = textbuf->length();
    text  = textbuf->text_range(start, end);
    style = stylebuf->text_range(start, end);

    style_parse(text, style, end - start);

    stylebuf->replace(start, end, style);
    ((Fl_Text_Editor *)cbArg)->redisplay_range(start, end);
  }

  free(text);
  free(style);
}
```

The `style_parse()` function scans a copy of the text in the buffer and generates the necessary style characters for display. It assumes that parsing begins at the start of a line:

```
//
// 'style_parse()' - Parse text and produce style data.
//

void
style_parse(const char *text,
            char       *style,
            int        length) {
  char             current;
  int             col;
  int             last;
  char             buf[255],
              bufptr;
  const char *temp;

  for (current = *style, col = 0, last = 0; length > 0; length --, text ++) {
    if (current == 'A') {
      // Check for directives, comments, strings, and keywords...
      if (col == 0 && *text == '#') {
        // Set style to directive
        current = 'E';
      } else if (strncmp(text, "//", 2) == 0) {
        current = 'B';
      } else if (strncmp(text, "/*", 2) == 0) {
        current = 'C';
      } else if (strncmp(text, "\\\"", 2) == 0) {
        // Quoted quote...
        *style++ = current;
        *style++ = current;
        text ++;
        length --;
        col += 2;
        continue;
      } else if (*text == '\"') {
        current = 'D';
      } else if (!last && islower(*text)) {
        // Might be a keyword...
        for (temp = text, bufptr = buf;
             islower(*temp) && bufptr < (buf + sizeof(buf) - 1);
             *bufptr++ = *temp++);

        if (!islower(*temp)) {
          *bufptr = '\0';
          bufptr = buf;

          if (bsearch(&bufptr, code_types,
                      sizeof(code_types) / sizeof(code_types[0]),
                      sizeof(code_types[0]), compare_keywords)) {
            while (text < temp) {
              *style++ = 'F';
              text ++;
              length --;
              col ++;
            }

            text --;
            length ++;
            last = 1;
            continue;
          } else if (bsearch(&bufptr, code_keywords,
                             sizeof(code_keywords) / sizeof(code_keywords[0]),
                             sizeof(code_keywords[0]), compare_keywords)) {
            while (text < temp) {
              *style++ = 'G';
              text ++;
              length --;
              col ++;
            }

            text --;
            length ++;
            last = 1;
            continue;
          }
        }
      }
    } else if (current == 'C' && strncmp(text, "*/", 2) == 0) {
      // Close a C comment...
       style++ = current;
       style++ = current;
      text ++;
      length --;
      current = 'A';
      col += 2;
      continue;
    } else if (current == 'D') {
      // Continuing in string...
      if (strncmp(text, "\\\"", 2) == 0) {
        // Quoted end quote...
         style++ = current;
         style++ = current;
        text ++;
        length --;
        col += 2;
        continue;
      } else if (*text == '\"') {
        // End quote...
         style++ = current;
        col ++;
        current = 'A';
        continue;
      }
    }

    // Copy style info...
    if (current == 'A' && (*text == '{' || *text == '}')) *style++ = 'G';
    else *style++ = current;
    col ++;

    last = isalnum(*text) || *text == '.';

    if (*text == '\n') {
      // Reset column and possibly reset the style
      col = 0;
      if (current == 'B' || current == 'E') current = 'A';
    }
  }
}
```

Chapter 7

Drawing Things in FLTK

This chapter covers the drawing functions that are provided with FLTK.

7.1 When Can You Draw Things in FLTK?

There are only certain places you can execute FLTK code that draws to the computer's display. Calling these functions at other places will result in undefined behavior!

- The most common place is inside the virtual Fl_Widget::draw() method. To write code here, you must subclass one of the existing Fl_Widget classes and implement your own version of draw().
- You can also create custom boxtypes and labeltypes. These involve writing small procedures that can be called by existing Fl_Widget::draw() methods. These "types" are identified by an 8-bit index that is stored in the widget's `box()`, `labeltype()`, and possibly other properties.
- You can call Fl_Window::make_current() to do incremental update of a widget. Use Fl_Widget::window() to find the window.

In contrast, code that draws to other drawing surfaces than the display (i.e., instances of derived classes of the Fl_Surface_Device class, except Fl_Display_Device, such as Fl_Printer and Fl_Copy_Surface) can be executed at any time as follows:

1. Memorize what is the current drawing surface calling Fl_Surface_Device::surface(), and make your surface the new current drawing surface calling the surface's set_current() function;
2. Make a series of calls to any of the drawing functions described below; these will operate on the new current drawing surface;
3. Set the current drawing surface back to its previous state calling the previous surface's set_current().

7.1.1 What Drawing Unit do FLTK drawing functions use?

When drawing to the display or to instances of Fl_Copy_Surface and Fl_Image_Surface, the unit of drawing functions corresponds generally to one pixel. The so-called 'retina' displays of some recent Apple computers are an exception to this rule: one drawing unit corresponds to the width or the height of 2 display pixels on a retina display.

When drawing to surfaces that are instances of Fl_Paged_Device derived classes (i.e., Fl_Printer or Fl_PostScript_File_Device), the drawing unit is initially one point, that is, 1/72 of an inch. But this unit is changed after calls to Fl_Paged_Device::scale().

7.2 Drawing Functions

To use the drawing functions you must first include the <FL/fl_draw.H> header file. FLTK provides the following types of drawing functions:

- Boxes
- Clipping
- Colors
- Line Dashes and Thickness
- Drawing Fast Shapes
- Drawing Complex Shapes
- Drawing Text
- Fonts
- Character Encoding
- Drawing Overlays
- Drawing Images
- Direct Image Drawing
- Direct Image Reading
- Image Classes
- Offscreen Drawing

7.2.1 Boxes

FLTK provides three functions that can be used to draw boxes for buttons and other UI controls. Each function uses the supplied upper-lefthand corner and width and height to determine where to draw the box.

void fl_draw_box(Fl_Boxtype b, int x, int y, int w, int h, Fl_Color c)

The `fl_draw_box()` function draws a standard boxtype `b` in the specified color `c`.

void fl_frame(const char *s, int x, int y, int w, int h)
void fl_frame2(const char *s, int x, int y, int w, int h)

The `fl_frame()` and `fl_frame2()` functions draw a series of line segments around the given box. The string `s` must contain groups of 4 letters which specify one of 24 standard grayscale values, where 'A' is black and 'X' is white. The results of calling these functions with a string that is not a multiple of 4 characters in length are undefined.

The only difference between `fl_frame()` and `fl_frame2()` is the order of the line segments:

- For `fl_frame()` the order of each set of 4 characters is: top, left, bottom, right.
- For `fl_frame2()` the order of each set of 4 characters is: bottom, right, top, left.

Note that fl_frame(Fl_Boxtype b) is described in the Box Types section.

7.2.2 Clipping

You can limit all your drawing to a rectangular region by calling `fl_push_clip()`, and put the drawings back by using `fl_pop_clip()`. This rectangle is measured in pixels and is unaffected by the current transformation matrix.

In addition, the system may provide clipping when updating windows which may be more complex than a simple rectangle.

void fl_push_clip(int x, int y, int w, int h)
void fl_clip(int x, int y, int w, int h)

Intersect the current clip region with a rectangle and push this new region onto the stack.

The `fl_clip()` version is deprecated and will be removed from future releases.

void fl_push_no_clip()

Pushes an empty clip region on the stack so nothing will be clipped.

void fl_pop_clip()

Restore the previous clip region.

Note: You must call `fl_pop_clip()` once for every time you call `fl_push_clip()`. If you return to FLTK with the clip stack not empty unpredictable results occur.

int fl_not_clipped(int x, int y, int w, int h)

Returns non-zero if any of the rectangle intersects the current clip region. If this returns 0 you don't have to draw the object.

Note: Under X this returns 2 if the rectangle is partially clipped, and 1 if it is entirely inside the clip region.

int fl_clip_box(int x, int y, int w, int h, int &X, int &Y, int &W, int &H)

Intersect the rectangle `x,y,w,h` with the current clip region and returns the bounding box of the result in `X,Y,W,H`. Returns non-zero if the resulting rectangle is different than the original. This can be used to limit the necessary drawing to a rectangle. `W` and `H` are set to zero if the rectangle is completely outside the region.

void fl_clip_region(Fl_Region r)
Fl_Region fl_clip_region()

Replace the top of the clip stack with a clipping region of any shape. Fl_Region is an operating system specific type. The second form returns the current clipping region.

7.3 Colors

FLTK manages colors as 32-bit unsigned integers, encoded as RGBI. When the "RGB" bytes are non-zero, the value is treated as RGB. If these bytes are zero, the "I" byte will be used as an index into the colormap. Colors with both "RGB" set and an "I" >0 are reserved for special use.

Values from 0 to 255, i.e. the "I" index value, represent colors from the FLTK 1.3.x standard colormap and are allocated as needed on screens without TrueColor support. The **Fl_Color** enumeration type defines the standard colors and color cube for the first 256 colors. All of these are named with symbols in <FL/-Enumerations.H>. Example:

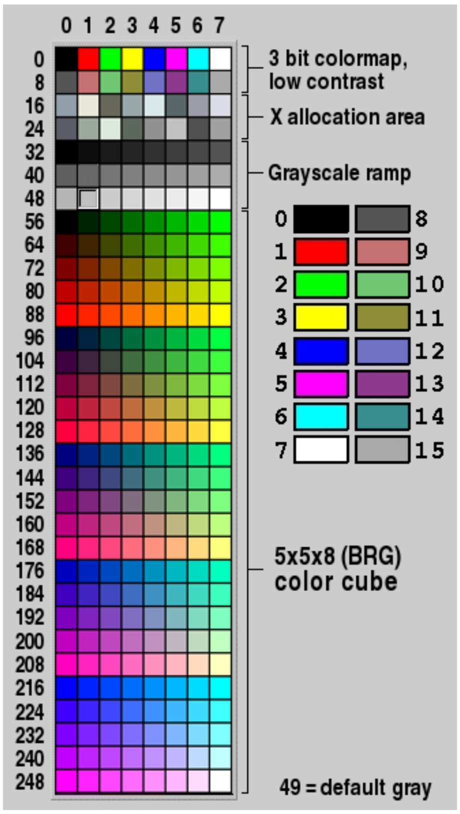


Figure 7.1: FLTK default colormap (Fl_Color 0x00 - 0xff)

Color values greater than 255 are treated as 24-bit RGB values. These are mapped to the closest color supported by the screen, either from one of the 256 colors in the FLTK 1.3.x colormap or a direct RGB value on TrueColor screens.

Fl_Color fl_rgb_color(uchar r, uchar g, uchar b)

Fl_Color fl_rgb_color(uchar grayscale)

Generate Fl_Color out of specified 8-bit RGB values or one 8-bit grayscale value.

void fl_color(Fl_Color c)

void fl_color(int c)

Sets the color for all subsequent drawing operations. Please use the first form: the second form is only provided for back compatibility.

For colormapped displays, a color cell will be allocated out of `fl_colormap` the first time you use a color. If the colormap fills up then a least-squares algorithm is used to find the closest color.

Fl_Color fl_color()

Returns the last color that was set using `fl_color()`. This can be used for state save/restore.

void fl_color(uchar r, uchar g, uchar b)

Set the color for all subsequent drawing operations. The closest possible match to the RGB color is used. The RGB color is used directly on TrueColor displays. For colormap visuals the nearest index in the gray ramp or color cube is used.

unsigned Fl::get_color(Fl_Color i)
void Fl::get_color(Fl_Color i, uchar &red, uchar &green, uchar &blue)

Generate RGB values from a colormap index value `i`. The first returns the RGB as a 32-bit unsigned integer, and the second decomposes the RGB into three 8-bit values.

Fl::get_system_colors()
Fl::foreground()
Fl::background()
Fl::background2()

The first gets color values from the user preferences or the system, and the other routines are used to apply those values.

Fl::own_colormap()
Fl::free_color(Fl_Color i, int overlay)
Fl::set_color(Fl_Color i, unsigned c)

`Fl::own_colormap()` is used to install a local colormap [X11 only].

`Fl::free_color()` and `Fl::set_color()` are used to remove and replace entries from the colormap.

There are two predefined graphical interfaces for choosing colors. The function fl_show_colormap() shows a table of colors and returns an Fl_Color index value. The Fl_Color_Chooser widget provides a standard RGB color chooser.

As the Fl_Color encoding maps to a 32-bit unsigned integer representing RGBI, it is also possible to specify a color using a hex constant as a color map index:

```
// COLOR MAP INDEX
color(0x000000II)
        ------ |
           |    |
           |   Color map index (8 bits)
           Must be zero
```

```
button->color(0x000000ff);                  // colormap index #255 (FL_WHITE)
```

or specify a color using a hex constant for the RGB components:

```
// RGB COLOR ASSIGNMENTS
color(0xRRGGBB00)
         | | | |
         | | | Must be zero
         | | Blue (8 bits)
         | Green (8 bits)
         Red (8 bits)
```

```
button->color(0xff000000);                   // RGB: red
button->color(0x00ff0000);                   // RGB: green
button->color(0x0000ff00);                   // RGB: blue
button->color(0xffffff00);                   // RGB: white
```

Note

If TrueColor is not available, any RGB colors will be set to the nearest entry in the colormap.

7.3.1 Line Dashes and Thickness

FLTK supports drawing of lines with different styles and widths. Full functionality is not available under Windows 95, 98, and Me due to the reduced drawing functionality these operating systems provide.

void fl_line_style(int style, int width, char∗ dashes)

Set how to draw lines (the "pen"). If you change this it is your responsibility to set it back to the default with `fl_line_style(0)`.

Note: Because of how line styles are implemented on MS Windows systems, you *must* set the line style *after* setting the drawing color. If you set the color after the line style you will lose the line style settings!

`style` is a bitmask which is a bitwise-OR of the following values. If you don't specify a dash type you will get a solid line. If you don't specify a cap or join type you will get a system-defined default of whatever value is fastest.

- `FL_SOLID       -------`
- `FL_DASH        - - - -`
- `FL_DOT         .......`
- `FL_DASHDOT     - .  - .`
- `FL_DASHDOTDOT  - ..  -`
- `FL_CAP_FLAT`
- `FL_CAP_ROUND`
- `FL_CAP_SQUARE` (extends past end point 1/2 line width)
- `FL_JOIN_MITER` (pointed)
- `FL_JOIN_ROUND`
- `FL_JOIN_BEVEL` (flat)

`width` is the number of pixels thick to draw the lines. Zero results in the system-defined default, which on both X and Windows is somewhat different and nicer than 1.

`dashes` is a pointer to an array of dash lengths, measured in pixels. The first location is how long to draw a solid portion, the next is how long to draw the gap, then the solid, etc. It is terminated with a zero-length entry. A `NULL` pointer or a zero-length array results in a solid line. Odd array sizes are not supported and result in undefined behavior.

Note: The dashes array does not work under Windows 95, 98, or Me, since those operating systems do not support complex line styles.

7.3.2 Drawing Fast Shapes

These functions are used to draw almost all the FLTK widgets. They draw on exact pixel boundaries and are as fast as possible. Their behavior is duplicated exactly on all platforms FLTK is ported. It is undefined whether these are affected by the transformation matrix, so you should only call these while the matrix is set to the identity matrix (the default).
void fl_point(int x, int y)

Draw a single pixel at the given coordinates.

void fl_rectf(int x, int y, int w, int h)
void fl_rectf(int x, int y, int w, int h, Fl_Color c)

Color a rectangle that exactly fills the given bounding box.

void fl_rectf(int x, int y, int w, int h, uchar r, uchar g, uchar b)

Color a rectangle with "exactly" the passed `r,g,b` color. On screens with less than 24 bits of color this is done by drawing a solid-colored block using fl_draw_image() so that the correct color shade is produced.

void fl_rect(int x, int y, int w, int h)
void fl_rect(int x, int y, int w, int h, Fl_Color c)

Draw a 1-pixel border *inside* this bounding box.

void fl_line(int x, int y, int x1, int y1)
void fl_line(int x, int y, int x1, int y1, int x2, int y2)

Draw one or two lines between the given points.

void fl_loop(int x, int y, int x1, int y1, int x2, int y2)
void fl_loop(int x, int y, int x1, int y1, int x2, int y2, int x3, int y3)

Outline a 3 or 4-sided polygon with lines.

void fl_polygon(int x, int y, int x1, int y1, int x2, int y2)
void fl_polygon(int x, int y, int x1, int y1, int x2, int y2, int x3, int y3)

Fill a 3 or 4-sided polygon. The polygon must be convex.

void fl_xyline(int x, int y, int x1)
void fl_xyline(int x, int y, int x1, int y2)
void fl_xyline(int x, int y, int x1, int y2, int x3)

Draw horizontal and vertical lines. A horizontal line is drawn first, then a vertical, then a horizontal.

void fl_yxline(int x, int y, int y1)
void fl_yxline(int x, int y, int y1, int x2)
void fl_yxline(int x, int y, int y1, int x2, int y3)

Draw vertical and horizontal lines. A vertical line is drawn first, then a horizontal, then a vertical.

void fl_arc(int x, int y, int w, int h, double a1, double a2)
void fl_pie(int x, int y, int w, int h, double a1, double a2)

Draw ellipse sections using integer coordinates. These functions match the rather limited circle drawing code provided by X and MS Windows. The advantage over using fl_arc() with floating point coordinates is that they are faster because they often use the hardware, and they draw much nicer small circles, since the small sizes are often hard-coded bitmaps.

If a complete circle is drawn it will fit inside the passed bounding box. The two angles are measured in degrees counter-clockwise from 3'oclock and are the starting and ending angle of the arc, `a2` must be greater or equal to `a1`.

`fl_arc()` draws a series of lines to approximate the arc. Notice that the integer version of `fl_arc()` has a different number of arguments to the other fl_arc() function described later in this chapter.

`fl_pie()` draws a filled-in pie slice. This slice may extend outside the line drawn by `fl_arc()`; to avoid this use `w-1` and `h-1`.

Todo add an Fl_Draw_Area_Cb typedef to allow fl_scroll(...) to be doxygenated?

void fl_scroll(int X, int Y, int W, int H, int dx, int dy, void (*draw_area)(void*, int,int,int,int), void* data)

Scroll a rectangle and draw the newly exposed portions. The contents of the rectangular area is first shifted by `dx` and `dy` pixels. The callback is then called for every newly exposed rectangular area,

7.3.3 Drawing Complex Shapes

The complex drawing functions let you draw arbitrary shapes with 2-D linear transformations. The functionality matches that found in the Adobe® PostScript™ language. The exact pixels that are filled are less defined than for the fast drawing functions so that FLTK can take advantage of drawing hardware. On both X and MS Windows the transformed vertices are rounded to integers before drawing the line segments: this severely limits the accuracy of these functions for complex graphics, so use OpenGL when greater accuracy and/or performance is required.
void fl_push_matrix()
void fl_pop_matrix()

Save and restore the current transformation. The maximum depth of the stack is 32 entries.

void fl_scale(double x,double y)
  void fl_scale(double x)
  void fl_translate(double x,double y)
  void fl_rotate(double d)
  void fl_mult_matrix(double a,double b,double c,double d,double x,double y)

Concatenate another transformation onto the current one. The rotation angle is in degrees (not radians) and is counter-clockwise.

double fl_transform_x(double x, double y)
  double fl_transform_y(double x, double y)
  double fl_transform_dx(double x, double y)
  double fl_transform_dy(double x, double y)
  void fl_transformed_vertex(double xf, double yf)

Transform a coordinate or a distance using the current transformation matrix. After transforming a coordinate pair, it can be added to the vertex list without any further translations using `fl_transformed_vertex`().

void fl_begin_points()
  void fl_end_points()

Start and end drawing a list of points. Points are added to the list with `fl_vertex`().

void fl_begin_line()
  void fl_end_line()

Start and end drawing lines.

void fl_begin_loop()
  void fl_end_loop()

Start and end drawing a closed sequence of lines.

void fl_begin_polygon()
  void fl_end_polygon()

Start and end drawing a convex filled polygon.

void fl_begin_complex_polygon()
  void fl_gap()
  void fl_end_complex_polygon()

Start and end drawing a complex filled polygon. This polygon may be concave, may have holes in it, or may be several disconnected pieces. Call `fl_gap`() to separate loops of the path. It is unnecessary but harmless to call `fl_gap`() before the first vertex, after the last one, or several times in a row.

`fl_gap`() should only be called between `fl_begin_complex_polygon`() and `fl_end_complex_polygon`(). To outline the polygon, use `fl_begin_loop`() and replace each `fl_gap`() with a `fl_end_loop`();fl_begin_loop() pair.

Note: For portability, you should only draw polygons that appear the same whether "even/odd" or "non-zero" winding rules are used to fill them. Holes should be drawn in the opposite direction of the outside loop.

void fl_vertex(double x,double y)

Add a single vertex to the current path.

void fl_curve(double X0, double Y0, double X1, double Y1, double X2, double Y2, double X3, double Y3)

Add a series of points on a Bezier curve to the path. The curve ends (and two of the points are) at `X0,Y0` and `X3,Y3`.

void fl_arc(double x, double y, double r, double start, double end)

Add a series of points to the current path on the arc of a circle; you can get elliptical paths by using scale and rotate before calling `fl_arc`(). The center of the circle is given by `x` and `y`, and `r` is its radius. `fl_arc`() takes `start` and `end` angles that are measured in degrees counter-clockwise from 3 o'clock. If `end` is less than `start` then it draws the arc in a clockwise direction.

void fl_circle(double x, double y, double r)

`fl_circle`(...) is equivalent to `fl_arc`(...,0,360) but may be faster. It must be the *only* thing in the path: if you want a circle as part of a complex polygon you must use `fl_arc`().

Note: `fl_circle`() draws incorrectly if the transformation is both rotated and non-square scaled.

7.3.4 Drawing Text

All text is drawn in the current font. It is undefined whether this location or the characters are modified by the current transformation.
void fl_draw(const char *, int x, int y)
void fl_draw(const char *, int n, int x, int y)

Draw a nul-terminated string or an array of `n` characters starting at the given location. Text is aligned to the left and to the baseline of the font. To align to the bottom, subtract `fl_descent`() from `y`. To align to the top, subtract `fl_descent`() and add `fl_height`(). This version of `fl_draw`() provides direct access to the text drawing function of the underlying OS. It does not apply any special handling to control characters.

void fl_draw(const char* str, int x, int y, int w, int h, Fl_Align align, Fl_Image* img, int draw_symbols)

Fancy string drawing function which is used to draw all the labels. The string is formatted and aligned inside the passed box. Handles '\t' and '\n', expands all other control characters to ^X, and aligns inside or against the edges of the box described by `x`, `y`, `w` and `h`. See Fl_Widget::align() for values for `align`. The value `FL_ALIGN_INSIDE` is ignored, as this function always prints inside the box.

If `img` is provided and is not `NULL`, the image is drawn above or below the text as specified by the `align` value.

The `draw_symbols` argument specifies whether or not to look for symbol names starting with the "@" character.

void fl_measure(const char *str, int& w, int& h, int draw_symbols)

Measure how wide and tall the string will be when printed by the `fl_draw`(...align) function. This includes leading/trailing white space in the string, kerning, etc.

If the incoming `w` is non-zero it will wrap to that width.

This will probably give unexpected values unless you have called fl_font() explicitly in your own code. Refer to the full documentation for fl_measure() for details on usage and how to avoid common pitfalls.

See Also

fl_text_extents() – measure the 'inked' area of a string
fl_width() – measure the pixel width of a string or single character
fl_height() – measure the pixel height of the current font
fl_descent() – the height of the descender for the current font

int fl_height()

Recommended minimum line spacing for the current font. You can also just use the value of `size` passed to fl_font().

See Also

fl_text_extents(), fl_measure(), fl_width(), fl_descent()

int fl_descent()

Recommended distance above the bottom of a `fl_height`() tall box to draw the text at so it looks centered vertically in that box.

double fl_width(const char* txt)
double fl_width(const char* txt, int n)
double fl_width(unsigned int unicode_char)

Return the pixel width of a nul-terminated string, a sequence of `n` characters, or a single character in the current font.

See Also

fl_measure(), fl_text_extents(), fl_height(), fl_descent()

void fl_text_extents(const char* txt, int& dx, int& dy, int& w, int& h)

Determines the minimum pixel dimensions of a nul-terminated string, ie. the 'inked area'.

Given a string "txt" drawn using fl_draw(txt, x, y) you would determine its pixel extents on the display using fl_text_extents(txt, dx, dy, wo, ho) such that a bounding box that exactly fits around the inked area of the text could be drawn with fl_rect(x+dx, y+dy, wo, ho).

Refer to the full documentation for fl_text_extents() for details on usage.

See Also

fl_measure(), fl_width(), fl_height(), fl_descent()

const char∗ fl_shortcut_label(int shortcut)

Unparse a shortcut value as used by Fl_Button or Fl_Menu_Item into a human-readable string like "-Alt+N". This only works if the shortcut is a character key or a numbered function key. If the shortcut is zero an empty string is returned. The return value points at a static buffer that is overwritten with each call.

7.3.5 Fonts

FLTK supports a set of standard fonts based on the Times, Helvetica/Arial, Courier, and Symbol typefaces, as well as custom fonts that your application may load. Each font is accessed by an index into a font table.

Initially only the first 16 faces are filled in. There are symbolic names for them: FL_HELVETICA, FL_TIMES, FL_COURIER, and modifier values FL_BOLD and FL_ITALIC which can be added to these, and FL_SYMBOL and FL_ZAPF_DINGBATS. Faces greater than 255 cannot be used in Fl_Widget labels, since Fl_Widget stores the index as a byte.

One important thing to note about 'current font' is that there are so many paths through the GUI event handling code as widgets are partially or completely hidden, exposed and then re-drawn and therefore you can not guarantee that 'current font' contains the same value that you set on the other side of the event loop. Your value may have been superseded when a widget was redrawn. You are strongly advised to set the font explicitly before you draw any text or query the width and height of text strings, etc.

void fl_font(int face, int size)

Set the current font, which is then used by the routines described above. You may call this outside a draw context if necessary to call fl_width(), but on X this will open the display.

The font is identified by a `face` and a `size`. The size of the font is measured in `pixels` and not "points". Lines should be spaced `size` pixels apart or more.

int fl_font()

int fl_size()

Returns the face and size set by the most recent call to `fl_font(a,b)`. This can be used to save/restore the font.

7.3.6 Character Encoding

FLTK 1.3 expects all text in Unicode UTF-8 encoding. UTF-8 is ASCII compatible for the first 128 characters. International characters are encoded in multibyte sequences.

FLTK expects individual characters, characters that are not part of a string, in UCS-4 encoding, which is also ASCII compatible, but requires 4 bytes to store a Unicode character.

For more information about character encodings, see the chapter on Unicode and UTF-8 Support.

7.3.7 Drawing Overlays

These functions allow you to draw interactive selection rectangles without using the overlay hardware. FLTK will XOR a single rectangle outline over a window.

void fl_overlay_rect(int x, int y, int w, int h)

void fl_overlay_clear()

`fl_overlay_rect()` draws a selection rectangle, erasing any previous rectangle by XOR'ing it first. `fl_overlay_clear()` will erase the rectangle without drawing a new one.

Using these functions is tricky. You should make a widget with both a `handle()` and `draw()` method. `draw()` should call `fl_overlay_clear()` before doing anything else. Your `handle()` method should call `window()->make_current()` and then `fl_overlay_rect()` after FL_DRAG events, and should call `fl_overlay_clear()` after a FL_RELEASE event.

7.4 Drawing Images

To draw images, you can either do it directly from data in your memory, or you can create a Fl_Image object. The advantage of drawing directly is that it is more intuitive, and it is faster if the image data changes more often than it is redrawn. The advantage of using the object is that FLTK will cache translated forms of the image (on X it uses a server pixmap) and thus redrawing is *much* faster.

7.4.1 Direct Image Drawing

The behavior when drawing images when the current transformation matrix is not the identity is not defined, so you should only draw images when the matrix is set to the identity.

void fl_draw_image(const uchar ∗buf,int X,int Y,int W,int H,int D,int L)
void fl_draw_image_mono(const uchar ∗buf,int X,int Y,int W,int H,int D,int L)

Draw an 8-bit per color RGB or luminance image. The pointer points at the "r" data of the top-left pixel. Color data must be in `r,g,b` order. The top left corner is given by `X` and `Y` and the size of the image is given by `W` and `H`. `D` is the delta to add to the pointer between pixels, it may be any value greater or equal to `3`, or it can be negative to flip the image horizontally. `L` is the delta to add to the pointer between lines (if 0 is passed it uses `W*D`). and may be larger than `W*D` to crop data, or negative to flip the image vertically.

It is highly recommended that you put the following code before the first show() of *any* window in your program to get rid of the dithering if possible:

```
Fl::visual(FL_RGB);
```

Gray scale (1-channel) images may be drawn. This is done if `abs(D)` is less than 3, or by calling `fl_draw_image_mono()`. Only one 8-bit sample is used for each pixel, and on screens with different numbers of bits for red, green, and blue only gray colors are used. Setting `D` greater than 1 will let you display one channel of a color image.

Note: The X version does not support all possible visuals. If FLTK cannot draw the image in the current visual it will abort. FLTK supports any visual of 8 bits or less, and all common TrueColor visuals up to 32 bits.

typedef void (∗Fl_Draw_Image_Cb)(void ∗data,int x,int y,int w,uchar ∗buf)
void fl_draw_image(Fl_Draw_Image_Cb cb,void ∗data,int X,int Y,int W,int H,int D)
void fl_draw_image_mono(Fl_Draw_Image_Cb cb,void ∗data,int X,int Y,int W,int H,int D)

Call the passed function to provide each scan line of the image. This lets you generate the image as it is being drawn, or do arbitrary decompression of stored data, provided it can be decompressed to individual scan lines easily.

The callback is called with the `void*` user data pointer which can be used to point at a structure of information about the image, and the `x`, `y`, and `w` of the scan line desired from the image. 0,0 is the upper-left corner of the image, *not* `X, Y`. A pointer to a buffer to put the data into is passed. You must copy `w` pixels from scanline `y`, starting at pixel `x`, to this buffer.

Due to cropping, less than the whole image may be requested. So `x` may be greater than zero, the first `y` may be greater than zero, and `w` may be less than `W`. The buffer is long enough to store the entire `W*D` pixels, this is for convenience with some decompression schemes where you must decompress the entire line at once: decompress it into the buffer, and then if `x` is not zero, copy the data over so the `x'th` pixel is at the start of the buffer.

You can assume the `y's` will be consecutive, except the first one may be greater than zero.

If `D` is 4 or more, you must fill in the unused bytes with zero.

int fl_draw_pixmap(char* const* data, int x, int y, Fl_Color bg)
int fl_draw_pixmap(const char* const* cdata, int x, int y, Fl_Color bg)

Draws XPM image data, with the top-left corner at the given position. The image is dithered on 8-bit displays so you won't lose color space for programs displaying both images and pixmaps. This function returns zero if there was any error decoding the XPM data.

To use an XPM, do:

```
#include "foo.xpm"
...
fl_draw_pixmap(foo, X, Y);
```

Transparent colors are replaced by the optional Fl_Color argument. To draw with true transparency you must use the Fl_Pixmap class.

int fl_measure_pixmap(char* const* data, int &w, int &h)
int fl_measure_pixmap(const char* const* cdata, int &w, int &h)

An XPM image contains the dimensions in its data. This function finds and returns the width and height. The return value is non-zero if the dimensions were parsed ok and zero if there was any problem.

7.4.2 Direct Image Reading

FLTK provides a single function for reading from the current window or off-screen buffer into a RGB(A) image buffer.
uchar* fl_read_image(uchar *p, int X, int Y, int W, int H, int alpha)

Read a RGB(A) image from the current window or off-screen buffer. The `p` argument points to a buffer that can hold the image and must be at least `W*H*3` bytes when reading RGB images and `W*H*4` bytes when reading RGBA images. If `NULL`, `fl_read_image()` will create an array of the proper size which can be freed using `delete[]`.

The `alpha` parameter controls whether an alpha channel is created and the value that is placed in the alpha channel. If 0, no alpha channel is generated.

7.4.3 Image Classes

FLTK provides a base image class called Fl_Image which supports creating, copying, and drawing images of various kinds, along with some basic color operations. Images can be used as labels for widgets using the `image()` and `deimage()` methods or drawn directly.

The Fl_Image class does almost nothing by itself, but is instead supported by three basic image types:

- Fl_Bitmap
- Fl_Pixmap
- Fl_RGB_Image

The Fl_Bitmap class encapsulates a mono-color bitmap image. The `draw()` method draws the image using the current drawing color.

The Fl_Pixmap class encapsulates a colormapped image. The `draw()` method draws the image using the colors in the file, and masks off any transparent colors automatically.

The Fl_RGB_Image class encapsulates a full-color (or grayscale) image with 1 to 4 color components. Images with an even number of components are assumed to contain an alpha channel that is used for transparency. The transparency provided by the draw() method is either a 24-bit blend against the existing window contents or a "screen door" transparency mask, depending on the platform and screen color depth.

char fl_can_do_alpha_blending()

`fl_can_do_alpha_blending()` will return 1, if your platform supports true alpha blending for RGBA images, or 0, if FLTK will use screen door transparency.

FLTK also provides several image classes based on the three standard image types for common file formats:

- Fl_GIF_Image
- Fl_JPEG_Image
- Fl_PNG_Image
- Fl_PNM_Image
- Fl_XBM_Image
- Fl_XPM_Image

Each of these image classes loads a named file of the corresponding format. The Fl_Shared_Image class can be used to load any type of image file - the class examines the file and constructs an image of the appropriate type. It can also be used to scale an image to a certain size in drawing units, independently from its size in pixels (see Fl_Shared_Image::scale()).

Finally, FLTK provides a special image class called Fl_Tiled_Image to tile another image object in the specified area. This class can be used to tile a background image in a Fl_Group widget, for example.

virtual void Fl_Image::copy()
virtual Fl_Image∗ Fl_Image::copy(int w, int h)

The `copy()` method creates a copy of the image. The second form specifies the new size of the image - the image is resized using the nearest-neighbor algorithm (this is the default).

Note

As of FLTK 1.3.3 the image resizing algorithm can be changed. See Fl_Image::RGB_scaling(Fl_RGB_Scaling method)

virtual void Fl_Image::draw(int x, int y, int w, int h, int ox, int oy)

The `draw()` method draws the image object. `x,y,w,h` indicates the destination rectangle. `ox,oy,w,h` is the source rectangle. This source rectangle is copied to the destination. The source rectangle may extend outside the image, i.e. `ox` and `oy` may be negative and `w` and `h` may be bigger than the image, and this area is left unchanged.

Note

See exceptions for Fl_Tiled_Image::draw() regarding arguments `ox`, `oy`, `w`, and `h`.

virtual void Fl_Image::draw(int x, int y)

Draws the image with the upper-left corner at `x, y`. This is the same as doing `img->draw(x, y, img->w(), img->h(), 0, 0)` where img is a pointer to any Fl_Image type.

7.4.4 Offscreen Drawing

Sometimes it can be very useful to generate a complex drawing in memory first and copy it to the screen at a later point in time. This technique can significantly reduce the amount of repeated drawing. Offscreen drawing functions are declared in <FL/x.H>. Fl_Double_Window uses offscreen rendering to avoid flickering on systems that don't support double-buffering natively.

Fl_Offscreen fl_create_offscreen(int w, int h)

Create an RGB offscreen buffer with `w*h` pixels.

void fl_delete_offscreen(Fl_Offscreen)

Delete a previously created offscreen buffer. All drawings are lost.

void fl_begin_offscreen(Fl_Offscreen)

Send all subsequent drawing commands to this offscreen buffer. FLTK can draw into a buffer at any time. There is no need to wait for an Fl_Widget::draw() to occur.

void fl_end_offscreen()

Quit sending drawing commands to this offscreen buffer.

void fl_copy_offscreen(int x, int y, int w, int h, Fl_Offscreen osrc, int srcx, int srcy)

Copy a rectangular area of the size `w*h` from `srcx`,srcy in the offscreen buffer into the current buffer at `x`,y.

Chapter 8

Handling Events

This chapter discusses the FLTK event model and how to handle events in your program or widget.

8.1 The FLTK Event Model

Every time a user moves the mouse pointer, clicks a button, or presses a key, an event is generated and sent to your application. Events can also come from other programs like the window manager.

Events are identified by the integer argument passed to a `handle()` method that overrides the Fl_Widget::handle() virtual method. Other information about the most recent event is stored in static locations and acquired by calling the Fl::event_*() methods. This static information remains valid until the next event is read from the window system, so it is ok to look at it outside of the `handle()` method.

Event numbers can be converted to their actual names using the fl_eventnames[] array defined in #include <FL/names.h>; see next chapter for details.

In the next chapter, the MyClass::handle() example shows how to override the Fl_Widget::handle() method to accept and process specific events.

8.2 Mouse Events

8.2.1 FL_PUSH

A mouse button has gone down with the mouse pointing at this widget. You can find out what button by calling Fl::event_button(). You find out the mouse position by calling Fl::event_x() and Fl::event_y().

A widget indicates that it *"wants"* the mouse click by returning non-zero from its `handle()` method, as in the MyClass::handle() example. It will then become the Fl::pushed() widget and will get `FL_DRAG` and the matching `FL_RELEASE` events. If `handle()` returns zero then FLTK will try sending the `FL_PUSH` to another widget.

8.2.2 FL_DRAG

The mouse has moved with a button held down. The current button state is in Fl::event_state(). The mouse position is in Fl::event_x() and Fl::event_y().

In order to receive `FL_DRAG` events, the widget must return non-zero when handling `FL_PUSH`.

8.2.3 FL_RELEASE

A mouse button has been released. You can find out what button by calling Fl::event_button().

In order to receive the `FL_RELEASE` event, the widget must return non-zero when handling `FL_PUSH`.

8.2.4 FL_MOVE

The mouse has moved without any mouse buttons held down. This event is sent to the Fl::belowmouse() widget.

In order to receive `FL_MOVE` events, the widget must return non-zero when handling `FL_ENTER`.

8.2.5 FL_MOUSEWHEEL

The user has moved the mouse wheel. The Fl::event_dx() and Fl::event_dy() methods can be used to find the amount to scroll horizontally and vertically.

8.3 Focus Events

8.3.1 FL_ENTER

The mouse has been moved to point at this widget. This can be used for highlighting feedback. If a widget wants to highlight or otherwise track the mouse, it indicates this by returning non-zero from its `handle()` method. It then becomes the Fl::belowmouse() widget and will receive `FL_MOVE` and `FL_LEAVE` events.

8.3.2 FL_LEAVE

The mouse has moved out of the widget.

In order to receive the `FL_LEAVE` event, the widget must return non-zero when handling `FL_ENTER`.

8.3.3 FL_FOCUS

This indicates an *attempt* to give a widget the keyboard focus.

If a widget wants the focus, it should change itself to display the fact that it has the focus, and return non-zero from its `handle()` method. It then becomes the Fl::focus() widget and gets `FL_KEYDOWN`, `FL_KEYUP`, and `FL_UNFOCUS` events.

The focus will change either because the window manager changed which window gets the focus, or because the user tried to navigate using tab, arrows, or other keys. You can check Fl::event_key() to figure out why it moved. For navigation it will be the key pressed and for interaction with the window manager it will be zero.

8.3.4 FL_UNFOCUS

This event is sent to the previous Fl::focus() widget when another widget gets the focus or the window loses focus.

8.4 Keyboard Events

8.4.1 FL_KEYBOARD, FL_KEYDOWN, FL_KEYUP

A key was pressed (FL_KEYDOWN) or released (FL_KEYUP). FL_KEYBOARD is a synonym for FL_-KEYDOWN, and both names are used interchangeably in this documentation.

The key can be found in Fl::event_key(). The text that the key should insert can be found with Fl::event-_text() and its length is in Fl::event_length().

If you use the key, then `handle()` should return 1. If you return zero then FLTK assumes you ignored the key and will then attempt to send it to a parent widget. If none of them want it, it will change the event into a `FL_SHORTCUT` event. FL_KEYBOARD events are also generated by the character palette/map.

To receive `FL_KEYBOARD` events you must also respond to the `FL_FOCUS` and `FL_UNFOCUS` events by returning 1. This way FLTK knows whether to bother sending your widget keyboard events. (Some widgets don't need them, e.g. Fl_Box.)

If you are writing a text-editing widget you may also want to call the Fl::compose() function to translate individual keystrokes into characters.

`FL_KEYUP` events are sent to the widget that currently has focus. This is not necessarily the same widget that received the corresponding `FL_KEYDOWN` event because focus may have changed between events.

Todo Add details on how to detect repeating keys, since on some X servers a repeating key will generate both FL_KEYUP and FL_KEYDOWN, such that to tell if a key is held, you need Fl::event_key(int) to detect if the key is being held down during FL_KEYUP or not.

8.4.2 FL_SHORTCUT

If the Fl::focus() widget is zero or ignores an `FL_KEYBOARD` event then FLTK tries sending this event to every widget it can, until one of them returns non-zero. `FL_SHORTCUT` is first sent to the Fl::belowmouse() widget, then its parents and siblings, and eventually to every widget in the window, trying to find an object that returns non-zero. FLTK tries really hard to not to ignore any keystrokes!

You can also make ”global” shortcuts by using Fl::add_handler(). A global shortcut will work no matter what windows are displayed or which one has the focus.

8.5 Widget Events

8.5.1 FL_DEACTIVATE

This widget is no longer active, due to deactivate() being called on it or one of its parents. Please note that although active() may still return true for this widget after receiving this event, it is only truly active if active() is true for both it and all of its parents. (You can use active_r() to check this).

8.5.2 FL_ACTIVATE

This widget is now active, due to activate() being called on it or one of its parents.

8.5.3 FL_HIDE

This widget is no longer visible, due to hide() being called on it or one of its parents, or due to a parent window being minimized. Please note that although visible() may still return true for this widget after receiving this event, it is only truly visible if visible() is true for both it and all of its parents. (You can use visible_r() to check this).

8.5.4 FL_SHOW

This widget is visible again, due to show() being called on it or one of its parents, or due to a parent window being restored. *A child Fl_Window will respond to this by actually creating the window if not done already, so if you subclass a window, be sure to pass `FL_SHOW` to the base class `handle()` method!*

Note

The events in this chapter (”Widget Events”), i.e. FL_ACTIVATE, FL_DEACTIVATE, FL_SHOW, and FL_HIDE, are the only events deactivated and invisible widgets can usually get, depending on their states. Under certain circumstances, there may also be FL_LEAVE or FL_UNFOCUS events delivered to deactivated or hidden widgets.

8.6 Clipboard Events

8.6.1 FL_PASTE

You should get this event some time after you call Fl::paste(). The contents of Fl::event_text() is the text to insert and the number of characters is in Fl::event_length().

8.6.2 FL_SELECTIONCLEAR

The Fl::selection_owner() will get this event before the selection is moved to another widget. This indicates that some other widget or program has claimed the selection. Motif programs used this to clear the selection indication. Most modern programs ignore this.

8.7 Drag and Drop Events

FLTK supports drag and drop of text and files from any application on the desktop to an FLTK widget. Text is transferred using UTF-8 encoding. Files are received as a list of full path and file names, separated by newline.

On some X11 platforms, files are received as a URL-encoded UTF-8 string, that is, non-ASCII bytes (and a few others such as space and %) are replaced by the 3 bytes "%XY" where XY are the byte's hexadecimal value. The fl_decode_uri() function can be used to transform in-place the received string into a proper UTF-8 string. On these platforms, strings corresponding to dropped files are further prepended by `file://` (or other prefixes such as `computer://`).

See Fl::dnd() for drag and drop from an FLTK widget.

The drag and drop data is available in Fl::event_text() at the concluding `FL_PASTE`. On some platforms, the event text is also available for the `FL_DND_*` events, however application must not depend on that behavior because it depends on the protocol used on each platform.

`FL_DND_*` events cannot be used in widgets derived from Fl_Group or Fl_Window.

8.7.1 FL_DND_ENTER

The mouse has been moved to point at this widget. A widget that is interested in receiving drag'n'drop data must return 1 to receive `FL_DND_DRAG`, `FL_DND_LEAVE` and `FL_DND_RELEASE` events.

8.7.2 FL_DND_DRAG

The mouse has been moved inside a widget while dragging data. A widget that is interested in receiving drag'n'drop data should indicate the possible drop position.

8.7.3 FL_DND_LEAVE

The mouse has moved out of the widget.

8.7.4 FL_DND_RELEASE

The user has released the mouse button dropping data into the widget. If the widget returns 1, it will receive the data in the immediately following `FL_PASTE` event.

8.8 Other events

8.8.1 FL_SCREEN_CONFIGURATION_CHANGED

Sent whenever the screen configuration changes (a screen is added/removed, a screen resolution is changed, screens are moved). Use Fl::add_handler() to be notified of this event.

8.8.2 FL_FULLSCREEN

The application window has been changed from normal to fullscreen, or from fullscreen to normal. If you are using a X window manager which supports Extended Window Manager Hints, this event will not be delivered until the change has actually happened.

8.9 Fl::event_*() methods

FLTK keeps the information about the most recent event in static storage. This information is good until the next event is processed. Thus it is valid inside `handle()` and `callback()` methods.

These are all trivial inline functions and thus very fast and small:

- Fl::event_button()
- Fl::event_clicks()
- Fl::event_dx()
- Fl::event_dy()
- Fl::event_inside()
- Fl::event_is_click()
- Fl::event_key()
- Fl::event_length()
- Fl::event_state()
- Fl::event_text()
- Fl::event_x()
- Fl::event_x_root()
- Fl::event_y()
- Fl::event_y_root()
- Fl::get_key()
- Fl::get_mouse()
- Fl::test_shortcut()

8.10 Event Propagation

Widgets receive events via the virtual handle() function. The argument indicates the type of event that can be handled. The widget must indicate if it handled the event by returning 1. FLTK will then remove the event and wait for further events from the host. If the widget's handle function returns 0, FLTK may redistribute the event based on a few rules.

Most events are sent directly to the `handle()` method of the Fl_Window that the window system says they belong to. The window (actually the Fl_Group that Fl_Window is a subclass of) is responsible for sending the events on to any child widgets. To make the Fl_Group code somewhat easier, FLTK sends some events (`FL_DRAG`, `FL_RELEASE`, `FL_KEYBOARD`, `FL_SHORTCUT`, `FL_UNFOCUS`, and `FL_LEAVE`) directly to leaf widgets. These procedures control those leaf widgets:

- Fl::add_handler()

- Fl::belowmouse()
- Fl::focus()
- Fl::grab()
- Fl::modal()
- Fl::pushed()
- Fl::release() (deprecated, see Fl::grab(0))
- Fl_Widget::take_focus()

FLTK propagates events along the widget hierarchy depending on the kind of event and the status of the UI. Some events are injected directly into the widgets, others may be resent as new events to a different group of receivers.

Mouse click events are first sent to the window that caused them. The window then forwards the event down the hierarchy until it reaches the widget that is below the click position. If that widget uses the given event, the widget is marked "pushed" and will receive all following mouse motion (FL_DRAG) events until the mouse button is released.

Mouse motion (FL_MOVE) events are sent to the Fl::belowmouse() widget, i.e. the widget that returned 1 on the last FL_ENTER event.

Mouse wheel events are sent to the window that caused the event. The window propagates the event down the tree, first to the widget that is below the mouse pointer, and if that does not succeed, to all other widgets in the group. This ensures that scroll widgets work as expected with the widget furthest down in the hierarchy getting the first opportunity to use the wheel event, but also giving scroll bars, that are not directly below the mouse a chance.

Keyboard events are sent directly to the widget that has keyboard focus. If the focused widget rejects the event, it is resent as a shortcut event, first to the top-most window, then to the widget below the mouse pointer, propagating up the hierarchy to all its parents. Those send the event also to all widgets that are not below the mouse pointer. Now if that did not work out, the shortcut is sent to all registered shortcut handlers.

If we are still unsuccessful, the event handler flips the case of the shortcut letter and starts over. Finally, if the key is "escape", FLTK sends a close event to the top-most window.

All other events are pretty much sent right away to the window that created the event.

Widgets can "grab" events. The grabbing window gets all events exclusively, but usually by the same rules as described above.

Windows can also request exclusivity in event handling by making the window modal.

8.11 FLTK Compose-Character Sequences

The character composition done by Fl_Input widget requires that you call the Fl::compose() function if you are writing your own text editor widget.

Currently, all characters made by single key strokes with or without modifier keys, or by system-defined character compose sequences (that can involve dead keys or a compose key) can be input. You should call Fl::compose() in case any enhancements to this processing are done in the future. The interface has been designed to handle arbitrary UTF-8 encoded text.

The following methods are provided for character composition:

- Fl::compose()
- Fl::compose_reset()

Under Mac OS X, FLTK "previews" partially composed sequences.

Chapter 9

Adding and Extending Widgets

This chapter describes how to add your own widgets or extend existing widgets in FLTK.

9.1 Subclassing

New widgets are created by *subclassing* an existing FLTK widget, typically Fl_Widget for controls and Fl_Group for composite widgets.

A control widget typically interacts with the user to receive and/or display a value of some sort.

A composite widget holds a list of child widgets and handles moving, sizing, showing, or hiding them as needed. Fl_Group is the main composite widget class in FLTK, and all of the other composite widgets (Fl_Pack, Fl_Scroll, Fl_Tabs, Fl_Tile, and Fl_Window) are subclasses of it.

You can also subclass other existing widgets to provide a different look or user-interface. For example, the button widgets are all subclasses of Fl_Button since they all interact with the user via a mouse button click. The only difference is the code that draws the face of the button.

9.2 Making a Subclass of Fl_Widget

Your subclasses can directly descend from Fl_Widget or any subclass of Fl_Widget. Fl_Widget has only four virtual methods, and overriding some or all of these may be necessary.

9.3 The Constructor

The constructor should have the following arguments:

```
MyClass(int x, int y, int w, int h, const char *label = 0);
```

This will allow the class to be used in FLUID without problems.

The constructor must call the constructor for the base class and pass the same arguments:

```
MyClass::MyClass(int x, int y, int w, int h, const char *label)
: Fl_Widget(x, y, w, h, label) {
// do initialization stuff...
}
```

Fl_Widget's protected constructor sets `x()`, `y()`, `w()`, `h()`, and `label()` to the passed values and initializes the other instance variables to:

```
type(0);
box(FL_NO_BOX);
color(FL_BACKGROUND_COLOR);
selection_color(FL_BACKGROUND_COLOR);
labeltype(FL_NORMAL_LABEL);
labelstyle(FL_NORMAL_STYLE);
```

```
labelsize(FL_NORMAL_SIZE);
labelcolor(FL_FOREGROUND_COLOR);
align(FL_ALIGN_CENTER);
callback(default_callback,0);
flags(ACTIVE|VISIBLE);
image(0);
deimage(0);
```

9.4 Protected Methods of Fl_Widget

The following methods are provided for subclasses to use:

- clear_visible()
- damage()
- draw_box()
- draw_focus()
- draw_label()
- set_flag()
- set_visible()
- test_shortcut()
- type()

void Fl_Widget::damage(uchar mask)
void Fl_Widget::damage(uchar mask, int x, int y, int w, int h)
uchar Fl_Widget::damage()

The first form indicates that a partial update of the object is needed. The bits in mask are OR'd into damage(). Your `draw()` routine can examine these bits to limit what it is drawing. The public method Fl_Widget::redraw() simply does `Fl_Widget::damage(FL_DAMAGE_ALL)`, but the implementation of your widget can call the public `damage(n)`.

The second form indicates that a region is damaged. If only these calls are done in a window (no calls to `damage(n)`) then FLTK will clip to the union of all these calls before drawing anything. This can greatly speed up incremental displays. The mask bits are OR'd into `damage()` unless this is a Fl_Window widget.

The third form returns the bitwise-OR of all `damage(n)` calls done since the last `draw()`.

When redrawing your widgets you should look at the damage bits to see what parts of your widget need redrawing. The `handle()` method can then set individual damage bits to limit the amount of drawing that needs to be done:

```
MyClass::handle(int event) {
  ...
  if (change_to_part1) damage(1);
  if (change_to_part2) damage(2);
  if (change_to_part3) damage(4);
}

MyClass::draw() {
  if (damage() & FL_DAMAGE_ALL) {
    ... draw frame/box and other static stuff ...
  }

  if (damage() & (FL_DAMAGE_ALL | 1)) draw_part1();
  if (damage() & (FL_DAMAGE_ALL | 2)) draw_part2();
  if (damage() & (FL_DAMAGE_ALL | 4)) draw_part3();
}
```

Todo Clarify Fl_Window::damage(uchar) handling - seems confused/wrong? ORing value doesn't match setting behaviour in FL_Widget.H!

void Fl_Widget::draw_box() const
void Fl_Widget::draw_box(Fl_Boxtype t, Fl_Color c) const

The first form draws this widget's `box()`, using the dimensions of the widget. The second form uses `t` as the box type and `c` as the color for the box.

void Fl_Widget::draw_focus()
void Fl_Widget::draw_focus(Fl_Boxtype t, int x, int y, int w, int h) const

Draws a focus box inside the widget's bounding box. The second form allows you to specify a different bounding box.

void Fl_Widget::draw_label() const
void Fl_Widget::draw_label(int x, int y, int w, int h) const
void Fl_Widget::draw_label(int x, int y, int w, int h, Fl_Align align) const

The first form is the usual function for a `draw()` method to call to draw the widget's label. It does not draw the label if it is supposed to be outside the box (on the assumption that the enclosing group will draw those labels).

The second form uses the passed bounding box instead of the widget's bounding box. This is useful so "centered" labels are aligned with some feature, like a moving slider.

The third form draws the label anywhere. It acts as though `FL_ALIGN_INSIDE` has been forced on so the label will appear inside the passed bounding box. This is designed for parent groups to draw labels with.

void Fl_Widget::set_flag(int c)

Calling `set_flag(SHORTCUT_LABEL)` modifies the behavior of draw_label() so that '&' characters cause an underscore to be printed under the next letter.

void Fl_Widget::set_visible()
void Fl_Widget::clear_visible()

Fast inline versions of Fl_Widget::hide() and Fl_Widget::show(). These do not send the `FL_HIDE` and `FL_SHOW` events to the widget.

int Fl_Widget::test_shortcut()
static int Fl_Widget::test_shortcut(const char ∗s)

The first version tests Fl_Widget::label() against the current event (which should be a `FL_SHORTCUT` event). If the label contains a '&' character and the character after it matches the keypress, this returns true. This returns false if the `SHORTCUT_LABEL` flag is off, if the label is `NULL`, or does not have a '&' character in it, or if the keypress does not match the character.

The second version lets you do this test against an arbitrary string.

Todo Clarify Fl_Widget::test_shortcut() explanations. Fl_Widget.h says Internal Use only, but subclassing chapter gives details!

uchar Fl_Widget::type() const
void Fl_Widget::type(uchar t)

The property Fl_Widget::type() can return an arbitrary 8-bit identifier, and can be set with the protected method `type(uchar t)`. This value had to be provided for Forms compatibility, but you can use it for any purpose you want. Try to keep the value less than 100 to not interfere with reserved values.

FLTK does not use RTTI (Run Time Typing Information) to enhance portability. But this may change in the near future if RTTI becomes standard everywhere.

If you don't have RTTI you can use the clumsy FLTK mechanism, by having `type()` use a unique value. These unique values must be greater than the symbol `FL_RESERVED_TYPE` (which is 100) and less than `FL_WINDOW` (unless you make a subclass of Fl_Window). Look through the header files for `FL_RESERVED_TYPE` to find an unused number. If you make a subclass of Fl_Window you must use `FL_WINDOW + n` (where `n` must be in the range 1 to 7).

9.5 Handling Events

The virtual method Fl_Widget::handle(int event) is called to handle each event passed to the widget. It can:

- Change the state of the widget.
- Call Fl_Widget::redraw() if the widget needs to be redisplayed.
- Call Fl_Widget::damage(uchar c) if the widget needs a partial-update (assuming you provide support for this in your draw() method).
- Call Fl_Widget::do_callback() if a callback should be generated.
- Call Fl_Widget::handle() on child widgets.

Events are identified by the integer argument. Other information about the most recent event is stored in static locations and acquired by calling the Fl::event_*() methods. This information remains valid until another event is handled.

Here is a sample `handle()` method for a widget that acts as a pushbutton and also accepts the keystroke `'x'` to cause the callback:

```
int MyClass::handle(int event) {
  switch(event) {
    case FL_PUSH:
      highlight = 1;
      redraw();
      return 1;
    case FL_DRAG: {
        int t = Fl::event_inside(this);
        if (t != highlight) {
          highlight = t;
          redraw();
        }
      }
      return 1;
    case FL_RELEASE:
      if (highlight) {
        highlight = 0;
        redraw();
        do_callback();
        // never do anything after a callback, as the callback
        // may delete the widget!
      }
      return 1;
    case FL_SHORTCUT:
      if (Fl::event_key() == 'x') {
        do_callback();
        return 1;
      }
      return 0;
    default:
      return Fl_Widget::handle(event);
  }
}
```

You must return non-zero if your `handle()` method uses the event. If you return zero, the parent widget will try sending the event to another widget.

For debugging purposes, event numbers can be printed as their actual event names using the fl_eventnames[] array, e.g.:

```
#include <FL/names.h>          // defines fl_eventnames[]
[..]
int MyClass::handle(int e) {
    printf("Event was %s (%d)\n", fl_eventnames[e], e);     // e.g. "Event was FL_PUSH (1)"
    [..]
```

9.6 Drawing the Widget

The `draw()` virtual method is called when FLTK wants you to redraw your widget. It will be called if and only if `damage()` is non-zero, and `damage()` will be cleared to zero after it returns. The `draw()` method should be declared protected so that it can't be called from non-drawing code.

The `damage()` value contains the bitwise-OR of all the `damage(n)` calls to this widget since it was last drawn. This can be used for minimal update, by only redrawing the parts whose bits are set. FLTK will turn on the `FL_DAMAGE_ALL` bit if it thinks the entire widget must be redrawn, e.g. for an expose event.

Expose events (and the damage(mask,x,y,w,h) function described above) will cause `draw()` to be called with FLTK's clipping turned on. You can greatly speed up redrawing in some cases by testing `fl_not_clipped(x,y,w,h)` or `fl_clip_box()` and skipping invisible parts.

Besides the protected methods described above, FLTK provides a large number of basic drawing functions, which are described in the chapter Drawing Things in FLTK.

9.7 Resizing the Widget

The `resize(x,y,w,h)` method is called when the widget is being resized or moved. The arguments are the new position, width, and height. `x()`, `y()`, `w()`, and `h()` still remain the old size. You must call `resize()` on your base class with the same arguments to get the widget size to actually change.

This should *not* call `redraw()`, at least if only the `x()` and `y()` change. This is because composite widgets like Fl_Scroll may have a more efficient way of drawing the new position.

9.8 Making a Composite Widget

A "composite" widget contains one or more "child" widgets. To make a composite widget you should subclass Fl_Group. It is possible to make a composite object that is not a subclass of Fl_Group, but you'll have to duplicate the code in Fl_Group anyways.

Instances of the child widgets may be included in the parent:

```
class MyClass : public Fl_Group {
  Fl_Button the_button;
  Fl_Slider the_slider;
  ...
};
```

The constructor has to initialize these instances. They are automatically added to the group, since the Fl_Group constructor does Fl_Group::begin(). *Don't forget to call Fl_Group::end() or use the Fl_End pseudo-class:*

```
MyClass::MyClass(int x, int y, int w, int h) :
  Fl_Group(x, y, w, h),
  the_button(x + 5, y + 5, 100, 20),
  the_slider(x, y + 50, w, 20)
{
  ...(you could add dynamically created child widgets here)...
  end(); // don't forget to do this!
}
```

The child widgets need callbacks. These will be called with a pointer to the children, but the widget itself may be found in the `parent()` pointer of the child. Usually these callbacks can be static private methods, with a matching private method:

```
void MyClass::static_slider_cb(Fl_Widget* v, void *) { // static method
  ((MyClass*)(v->parent())->slider_cb();
}
void MyClass::slider_cb() { // normal method
  use(the_slider->value());
}
```

If you make the `handle()` method, you can quickly pass all the events to the children using the Fl_Group::handle() method. You don't need to override `handle()` if your composite widget does nothing other than pass events to the children:

```
int MyClass::handle(int event) {
  if (Fl_Group::handle(event)) return 1;
  ... handle events that children don't want ...
}
```

If you override `draw()` you need to draw all the children. If `redraw()` or `damage()` is called on a child, `damage(FL_DAMAGE_CHILD)` is done to the group, so this bit of `damage()` can be used to indicate that a child needs to be drawn. It is fastest if you avoid drawing anything else in this case:

```
int MyClass::draw() {
  Fl_Widget *const*a = array();
  if (damage() == FL_DAMAGE_CHILD) { // only redraw some children
    for (int i = children(); i --; a ++) update_child(**a);
  } else { // total redraw
    ... draw background graphics ...
    // now draw all the children atop the background:
    for (int i = children_; i --; a ++) {
      draw_child(**a);
      draw_outside_label(**a); // you may not need to do this
    }
  }
}
```

Fl_Group provides some protected methods to make drawing easier:

- draw_child()
- draw_children()
- draw_outside_label()
- update_child()

void Fl_Group::draw_child(Fl_Widget &widget) const

This will force the child's `damage()` bits all to one and call `draw()` on it, then clear the `damage()`. You should call this on all children if a total redraw of your widget is requested, or if you draw something (like a background box) that damages the child. Nothing is done if the child is not `visible()` or if it is clipped.

void Fl_Group::draw_children()

A convenience function that draws all children of the group. This is useful if you derived a widget from Fl_Group and want to draw a special border or background. You can call `draw_children()` from the derived `draw()` method after drawing the box, border, or background.

void Fl_Group::draw_outside_label(const Fl_Widget &widget) const

Draw the labels that are *not* drawn by draw_label(). If you want more control over the label positions you might want to call `child->draw_label(x,y,w,h,a)`.

void Fl_Group::update_child(Fl_Widget& widget) const

Draws the child only if its `damage()` is non-zero. You should call this on all the children if your own damage is equal to `FL_DAMAGE_CHILD`. Nothing is done if the child is not `visible()` or if it is clipped.

9.9 Cut and Paste Support

FLTK provides routines to cut and paste UTF-8 encoded text between applications:

- Fl::copy()
- Fl::paste()
- Fl::selection()
- Fl::selection_owner()

It is also possible to copy and paste image data between applications:

- Fl_Copy_Surface
- Fl::clipboard_contains()
- Fl::paste()

It may be possible to cut/paste other kinds of data by using Fl::add_handler(). Note that handling events beyond those provided by FLTK may be operating system specific. See Operating System Issues for more details.

9.10 Drag And Drop Support

FLTK provides routines to drag and drop UTF-8 encoded text between applications:

Drag'n'drop operations are initiated by copying data to the clipboard and calling the function Fl::dnd().

Drop attempts are handled via the following events, already described under Drag and Drop Events in a previous chapter:

- `FL_DND_ENTER`

- `FL_DND_DRAG`
- `FL_DND_LEAVE`
- `FL_DND_RELEASE`
- `FL_PASTE`

9.11 Making a subclass of Fl_Window

You may want your widget to be a subclass of Fl_Window, Fl_Double_Window, or Fl_Gl_Window. This can be useful if your widget wants to occupy an entire window, and can also be used to take advantage of system-provided clipping, or to work with a library that expects a system window ID to indicate where to draw.

Subclassing Fl_Window is almost exactly like subclassing Fl_Group, and in fact you can easily switch a subclass back and forth. Watch out for the following differences:

1. Fl_Window is a subclass of Fl_Group so *make sure your constructor calls* `end()` unless you actually want children added to your window.
2. When handling events and drawing, the upper-left corner is at 0,0, not `x(),y()` as in other Fl_-Widget's. For instance, to draw a box around the widget, call `draw_box(0,0,w(),h())`, rather than `draw_box(x(),y(),w(),h())`.

You may also want to subclass Fl_Window in order to get access to different visuals or to change other attributes of the windows. See the Operating System Issues chapter for more information.

Chapter 10

Using OpenGL

This chapter discusses using FLTK for your OpenGL applications.

10.1 Using OpenGL in FLTK

The easiest way to make an OpenGL display is to subclass Fl_Gl_Window. Your subclass must implement a `draw()` method which uses OpenGL calls to draw the display. Your main program should call `redraw()` when the display needs to change, and (somewhat later) FLTK will call `draw()`.

With a bit of care you can also use OpenGL to draw into normal FLTK windows. This allows you to use Gouraud shading for drawing your widgets. To do this you use the gl_start() and gl_finish() functions around your OpenGL code.

You must include FLTK's <`FL/gl.h`> header file. It will include the file <`GL/gl.h`>, define some extra drawing functions provided by FLTK, and include the <`windows.h`> header file needed by WIN32 applications.

Some simple coding rules (see OpenGL and 'retina' displays) allow to write cross-platform code that will draw high resolution OpenGL graphics if run on 'retina' displays with Mac OS X.

10.2 Making a Subclass of Fl_Gl_Window

To make a subclass of Fl_Gl_Window, you must provide:

- A class definition.
- A `draw()` method.
- A `handle()` method if you need to receive input from the user.

If your subclass provides static controls in the window, they must be redrawn whenever the `FL_DAMAGE_ALL` bit is set in the value returned by `damage()`. For double-buffered windows you will need to surround the drawing code with the following code to make sure that both buffers are redrawn:

```
#ifndef MESA
glDrawBuffer(GL_FRONT_AND_BACK);
#endif // !MESA
... draw stuff here ...
#ifndef MESA
glDrawBuffer(GL_BACK);
#endif // !MESA
```

Note:
If you are using the Mesa graphics library, the call to `glDrawBuffer()` is not required and will slow down drawing considerably. The preprocessor instructions shown above will optimize your code based upon the graphics library used.

10.2.1 Defining the Subclass

To define the subclass you just subclass the Fl_Gl_Window class:

```
class MyWindow : public Fl_Gl_Window {
  void draw();
  int handle(int);

public:
  MyWindow(int X, int Y, int W, int H, const char *L)
    : Fl_Gl_Window(X, Y, W, H, L) {}
};
```

The `draw()` and `handle()` methods are described below. Like any widget, you can include additional private and public data in your class (such as scene graph information, etc.)

10.2.2 The draw() Method

The `draw()` method is where you actually do your OpenGL drawing:

```
void MyWindow::draw() {
  if (!valid()) {
    ... set up projection, viewport, etc ...
    ... window size is in w() and h().
    ... valid() is turned on by FLTK after draw() returns
  }
  ... draw ...
}
```

10.2.3 The handle() Method

The `handle()` method handles mouse and keyboard events for the window:

```
int MyWindow::handle(int event) {
  switch(event) {
  case FL_PUSH:
    ... mouse down event ...
    ... position in Fl::event_x() and Fl::event_y()
    return 1;
  case FL_DRAG:
    ... mouse moved while down event ...
    return 1;
  case FL_RELEASE:
    ... mouse up event ...
    return 1;
  case FL_FOCUS :
  case FL_UNFOCUS :
    ... Return 1 if you want keyboard events, 0 otherwise
    return 1;
  case FL_KEYBOARD:
    ... keypress, key is in Fl::event_key(), ascii in Fl::event_text()
    ... Return 1 if you understand/use the keyboard event, 0 otherwise...
    return 1;
  case FL_SHORTCUT:
    ... shortcut, key is in Fl::event_key(), ascii in Fl::event_text()
    ... Return 1 if you understand/use the shortcut event, 0 otherwise...
    return 1;
  default:
    // pass other events to the base class...
    return Fl_Gl_Window::handle(event);
  }
}
```

When `handle()` is called, the OpenGL context is not set up! If your display changes, you should call `redraw()` and let `draw()` do the work. Don't call any OpenGL drawing functions from inside `handle()`!

You can call *some* OpenGL stuff like hit detection and texture loading functions by doing:

```
case FL_PUSH:
  make_current();     // make OpenGL context current
  if (!valid()) {

    ... set up projection exactly the same as draw ...

    valid(1);         // stop it from doing this next time
  }
  ... ok to call NON-DRAWING OpenGL code here, such as hit
  detection, loading textures, etc...
```

Your main program can now create one of your windows by doing `new MyWindow(...)`.

You can also use your new window class in FLUID by:

1. Putting your class definition in a `MyWindow.H` file.
2. Creating a Fl_Box widget in FLUID.
3. In the widget panel fill in the "class" field with `MyWindow`. This will make FLUID produce constructors for your new class.
4. In the "Extra Code" field put `#include "MyWindow.H"`, so that the FLUID output file will compile.

You must put `glwindow->show()` in your main code after calling `show()` on the window containing the OpenGL window.

10.3 Using OpenGL in Normal FLTK Windows

You can put OpenGL code into the `draw()` method, as described in Drawing the Widget in the previous chapter, or into the code for a boxtype or other places with some care.

Most importantly, before you show *any* windows, including those that don't have OpenGL drawing, you **must** initialize FLTK so that it knows it is going to use OpenGL. You may use any of the symbols described for `Fl_Gl_Window::mode()` to describe how you intend to use OpenGL:

```
Fl::gl_visual(FL_RGB);
```

You can then put OpenGL drawing code anywhere you can draw normally by surrounding it with gl_start() and gl_finish() to set up, and later release, an OpenGL context with an orthographic projection so that 0,0 is the lower-left corner of the window and each pixel is one unit. The current clipping is reproduced with OpenGL `glScissor()` commands. These functions also synchronize the OpenGL graphics stream with the drawing done by other X, WIN32, or FLTK functions.

```
gl_start();
... put your OpenGL code here ...
gl_finish();
```

The same context is reused each time. If your code changes the projection transformation or anything else you should use `glPushMatrix()` and `glPopMatrix()` functions to put the state back before calling `gl_finish()`.

You may want to use `Fl_Window::current()->h()` to get the drawable height so that you can flip the Y coordinates.

Unfortunately, there are a bunch of limitations you must adhere to for maximum portability:

- You must choose a default visual with Fl::gl_visual().
- You cannot pass `FL_DOUBLE` to Fl::gl_visual().
- You cannot use Fl_Double_Window or Fl_Overlay_Window.

Do *not* call `gl_start()` or `gl_finish()` when drawing into an Fl_Gl_Window !

10.4 OpenGL Drawing Functions

FLTK provides some useful OpenGL drawing functions. They can be freely mixed with any OpenGL calls, and are defined by including `<FL/gl.h>` which you should include instead of the OpenGL header `<GL/gl.h>`.

void gl_color(Fl_Color)

Sets the current OpenGL color to a FLTK color. *For color-index modes it will use `fl_xpixel(c)`, which is only right if this window uses the default colormap!*

void gl_rect(int x, int y, int w, int h)
void gl_rectf(int x, int y, int w, int h)

Outlines or fills a rectangle with the current color. If Fl_Gl_Window::ortho() has been called, then the rectangle will exactly fill the pixel rectangle passed.

void gl_font(Fl_Font fontid, int size)

Sets the current OpenGL font to the same font you get by calling fl_font().

int gl_height()
int gl_descent()
float gl_width(const char ∗s)
float gl_width(const char ∗s, int n)
float gl_width(uchar c)

Returns information about the current OpenGL font.

void gl_draw(const char ∗s)
void gl_draw(const char ∗s, int n)

Draws a nul-terminated string or an array of `n` characters in the current OpenGL font at the current raster position.

void gl_draw(const char ∗s, int x, int y)
void gl_draw(const char ∗s, int n, int x, int y)
void gl_draw(const char ∗s, float x, float y)
void gl_draw(const char ∗s, int n, float x, float y)

Draws a nul-terminated string or an array of `n` characters in the current OpenGL font at the given position.

void gl_draw(const char ∗s, int x, int y, int w, int h, Fl_Align)

Draws a string formatted into a box, with newlines and tabs expanded, other control characters changed to ^X, and aligned with the edges or center. Exactly the same output as fl_draw().

10.5 Speeding up OpenGL

Performance of Fl_Gl_Window may be improved on some types of OpenGL implementations, in particular MESA and other software emulators, by setting the `GL_SWAP_TYPE` environment variable. This variable declares what is in the backbuffer after you do a swapbuffers.

- `setenv GL_SWAP_TYPE COPY`

  This indicates that the back buffer is copied to the front buffer, and still contains its old data. This is true of many hardware implementations. Setting this will speed up emulation of overlays, and widgets that can do partial update can take advantage of this as `damage()` will not be cleared to -1.

- `setenv GL_SWAP_TYPE NODAMAGE`

  This indicates that nothing changes the back buffer except drawing into it. This is true of MESA and Win32 software emulation and perhaps some hardware emulation on systems with lots of memory.

- All other values for `GL_SWAP_TYPE`, and not setting the variable, cause FLTK to assume that the back buffer must be completely redrawn after a swap.

This is easily tested by running the gl_overlay demo program and seeing if the display is correct when you drag another window over it or if you drag the window off the screen and back on. You have to exit and run the program again for it to see any changes to the environment variable.

10.6 Using OpenGL Optimizer with FLTK

`OpenGL Optimizer` is a scene graph toolkit for OpenGL available from Silicon Graphics for IRIX and Microsoft Windows. It allows you to view large scenes without writing a lot of OpenGL code.

OptimizerWindow Class Definition

To use `OpenGL Optimizer` with FLTK you'll need to create a subclass of Fl_Gl_Widget that includes several state variables:

```
class OptimizerWindow : public Fl_Gl_Window {
  csContext *context_; // Initialized to 0 and set by draw()...
  csDrawAction *draw_action_; // Draw action...
  csGroup *scene_; // Scene to draw...
  csCamara *camera_; // Viewport for scene...

  void draw();

public:
  OptimizerWindow(int X, int Y, int W, int H, const char *L)
    : Fl_Gl_Window(X, Y, W, H, L) {
      context_ = (csContext *)0;
      draw_action_ = (csDrawAction *)0;
      scene_ = (csGroup *)0;
      camera_ = (csCamera *)0;
    }

  void scene(csGroup *g) { scene_ = g; redraw(); }

  void camera(csCamera *c) {
    camera_ = c;
    if (context_) {
      draw_action_->setCamera(camera_);
      camera_->draw(draw_action_);
      redraw();
    }
  }
};
```

The camera() Method

The camera() method sets the camera (projection and viewpoint) to use when drawing the scene. The scene is redrawn after this call.

The draw() Method

The draw() method performs the needed initialization and does the actual drawing:

```
void OptimizerWindow::draw() {
  if (!context_) {
    // This is the first time we've been asked to draw; create the
    // Optimizer context for the scene...

#ifdef WIN32
    context_ = new csContext((HDC)fl_getHDC());
    context_->ref();
    context_->makeCurrent((HDC)fl_getHDC());
#else
    context_ = new csContext(fl_display, fl_visual);
    context_->ref();
    context_->makeCurrent(fl_display, fl_window);
#endif // WIN32

    ... perform other context setup as desired ...

    // Then create the draw action to handle drawing things...

    draw_action_ = new csDrawAction;
    if (camera_) {
      draw_action_->setCamera(camera_);
      camera_->draw(draw_action_);
    }
  } else {
#ifdef WIN32
    context_->makeCurrent((HDC)fl_getHDC());
#else
    context_->makeCurrent(fl_display, fl_window);
#endif // WIN32
  }

  if (!valid()) {
    // Update the viewport for this context...
    context_->setViewport(0, 0, w(), h());
  }

  // Clear the window...
  context_->clear(csContext::COLOR_CLEAR | csContext::DEPTH_CLEAR,
                  0.0f,         // Red
                  0.0f,         // Green
                  0.0f,         // Blue
                  1.0f);        // Alpha

  // Then draw the scene (if any)...
  if (scene_)
    draw_action_->apply(scene_);
}
```

The scene() Method

The scene() method sets the scene to be drawn. The scene is a collection of 3D objects in a csGroup. The scene is redrawn after this call.

10.7 Using OpenGL 3.0 (or higher versions)

The examples subdirectory contains OpenGL3test.cxx, a toy program showing how to use OpenGL 3.0 (or higher versions) with FLTK in a cross-platform fashion. It contains also OpenGL3-glut-test.cxx which shows how to use FLTK's GLUT compatibility and OpenGL 3.

On the MSWindows and Unix/Linux platforms, FLTK creates contexts implementing the highest OpenGL version supported by the hardware, which are also compatible with lower OpenGL versions. Thus, FLTK allows source code targeting any version of OpenGL. Access to functions from OpenGL versions above 1.1 requires to load function pointers at runtime on these platforms. FLTK recommends to use the GLEW library to perform this. It is therefore necessary to install the GLEW library (see below). **On the Mac OS X platform**, FLTK creates by default contexts implementing OpenGL versions 1 or 2. To access OpenGL 3.0 (or higher versions), use the `FL_OPENGL3` flag (see below). Mac OS 10.7 or above is required; GLEW is possible but not necessary.

GLEW installation (Unix/Linux and MSWindows platforms)

GLEW is available as a package for most Linux distributions and in source form at `http://glew.sourceforge.net/`. For the MSWindows platform, a Visual Studio static library (glew32.lib) can be downloaded from the same web site; a MinGW-style static library (libglew32.a) can be built from source with the make command.

Source-level changes for OpenGL 3:

- Put this in all OpenGL-using source files (instead of #include <FL/gl.h>, and before #include <FL/glut.h> if you use GLUT):

```
#if defined(__APPLE__)
#  include <OpenGL/gl3.h> // defines OpenGL 3.0+ functions
#else
#  if defined(WIN32)
#    define GLEW_STATIC 1
#  endif
#  include <GL/glew.h>
#endif
```

- Add the `FL_OPENGL3` flag when calling Fl_Gl_Window::mode(int a) or glutInitDisplayMode().
- Put this in the `handle(int event)` member function of the first to be created among your Fl_Gl_Window-derived classes:

```
#ifndef __APPLE__
    static int first = 1;
    if (first && event == FL_SHOW && shown()) {
      first = 0;
      make_current();
      glewInit(); // defines pters to functions of OpenGL V 1.2 and above
    }
#endif
```

- Alternatively, if you use GLUT, put

```
#ifndef __APPLE__
  glewInit(); // defines pters to functions of OpenGL V 1.2 and above
#endif
```

  after the first glutCreateWindow() call.

If GLEW is installed on the Mac OS development platform, it is possible to use the same code for all platforms, with one exception: put

```
#ifdef __APPLE__
glewExperimental = GL_TRUE;
#endif
```

before the glewInit() call.

Changes in the build process

Link with libGLEW.so (on Unix/Linux), libglew32.a (with MinGW) or glew32.lib (with MS Visual Studio); no change is needed on the Mac OS platform.

Chapter 11

Programming with FLUID

This chapter shows how to use the Fast Light User-Interface Designer ("FLUID") to create your GUIs.

Subchapters:

11.1 What is FLUID?

The Fast Light User Interface Designer, or FLUID, is a graphical editor that is used to produce FLTK source code. FLUID edits and saves its state in `.fl` files. These files are text, and you can (with care) edit them in a text editor, perhaps to get some special effects.

FLUID can "compile" the `.fl` file into a `.cxx` and a `.h` file. The `.cxx` file defines all the objects from the `.fl` file and the `.h` file declares all the global ones. FLUID also supports localization (Internationalization) of label strings using message files and the GNU gettext or POSIX catgets interfaces.

A simple program can be made by putting all your code (including a `main()` function) into the `.fl` file and thus making the `.cxx` file a single source file to compile. Most programs are more complex than this, so you write other `.cxx` files that call the FLUID functions. These `.cxx` files must `#include` the `.h` file or they can `#include` the `.cxx` file so it still appears to be a single source file.

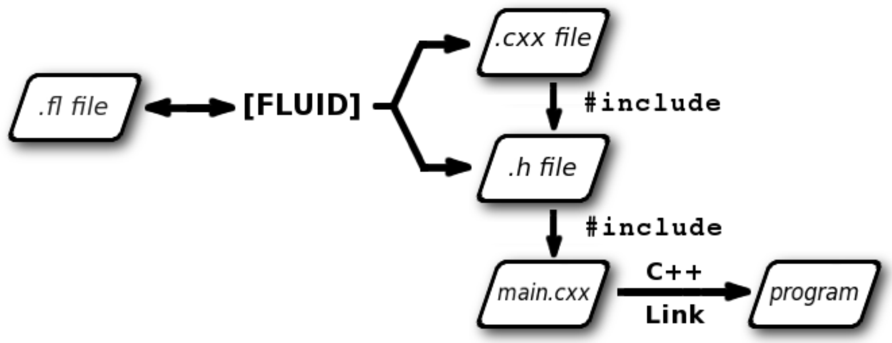


Figure 11.1: FLUID organization

Normally the FLUID file defines one or more functions or classes which output C++ code. Each function defines one or more FLTK windows, and all the widgets that go inside those windows.

Widgets created by FLUID are either ”named”, ”complex named” or ”unnamed”. A named widget has a legal C++ variable identifier as its name (i.e. only alphanumeric and underscore). In this case FLUID defines a global variable or class member that will point at the widget after the function defining it is called. A complex named object has punctuation such as `'.'` or `'->'` or any other symbols in its name. In this case FLUID assigns a pointer to the widget to the name, but does not attempt to declare it. This can be used to get the widgets into structures. An unnamed widget has a blank name and no pointer is stored.

Widgets may either call a named callback function that you write in another source file, or you can supply a small piece of C++ source and FLUID will write a private callback function into the `.cxx` file.

11.2 Running FLUID Under UNIX

To run FLUID under UNIX, type:

```
fluid filename.fl &
```

to edit the `.fl` file `filename.fl`. If the file does not exist you will get an error pop-up, but if you dismiss it you will be editing a blank file of that name. You can run FLUID without any name, in which case you will be editing an unnamed blank setup (but you can use save-as to write it to a file).

You can provide any of the standard FLTK switches before the filename:

```
-display host:n.n
-geometry WxH+X+Y
-title windowtitle
-name classname
-iconic
-fg color
-bg color
-bg2 color
-scheme schemename
```

Changing the colors may be useful to see what your interface will look at if the user calls it with the same switches. Similarly, using ”-scheme plastic” will show how the interface will look using the ”plastic” scheme.

In the current version, if you don’t put FLUID into the background with `'&'` then you will be able to abort FLUID by typing `CTRL-C` on the terminal. It will exit immediately, losing any changes.

11.3 Running FLUID Under Microsoft Windows

To run FLUID under WIN32, double-click on the *FLUID.exe* file. You can also run FLUID from the Command Prompt window. FLUID always runs in the background under WIN32.

11.4 Compiling .fl files

FLUID can also be called as a command-line "compiler" to create the `.cxx` and `.h` file from a `.fl` file. To do this type:

```
fluid -c filename.fl
```

This is the same as the menu 'File/Write Code...'. It will read the `filename.fl` file and write `filename.cxx` and `filename.h`. Any leading directory on `filename.fl` will be stripped, so they are always written to the current directory. If there are any errors reading or writing the files, FLUID will print the error and exit with a non-zero code. You can use the following lines in a makefile to automate the creation of the source and header files:

```
my_panels.h my_panels.cxx: my_panels.fl
        fluid -c my_panels.fl
```

Most versions of make support rules that cause `.fl` files to be compiled:

```
.SUFFIXES: .fl .cxx .h
.fl.h .fl.cxx:
        fluid -c $<
```

If you use

```
fluid -cs filename.fl
```

FLUID will also write the "strings" for internationalization in file 'filename.txt' (menu: 'File/Write Strings...').

Finally there is another option which is useful for program developers who have many .fl files and want to upgrade them to the current FLUID version. FLUID will read the `filename.fl` file, save it, and exit immediately. This writes the file with current syntax and options and the current FLTK version in the header of the file. Use

```
fluid -u filename.fl
```

to 'upgrade' `filename.fl` . You may combine this with '-c' or '-cs'.

Note

> All these commands overwrite existing files w/o warning. You should particularly take care when running 'fluid -u' since this overwrites the original .fl source file.

11.5 A Short Tutorial

FLUID is an amazingly powerful little program. However, this power comes at a price as it is not always obvious how to accomplish seemingly simple tasks with it. This tutorial will show you how to generate a complete user interface class with FLUID that is used for the CubeView program provided with FLTK.

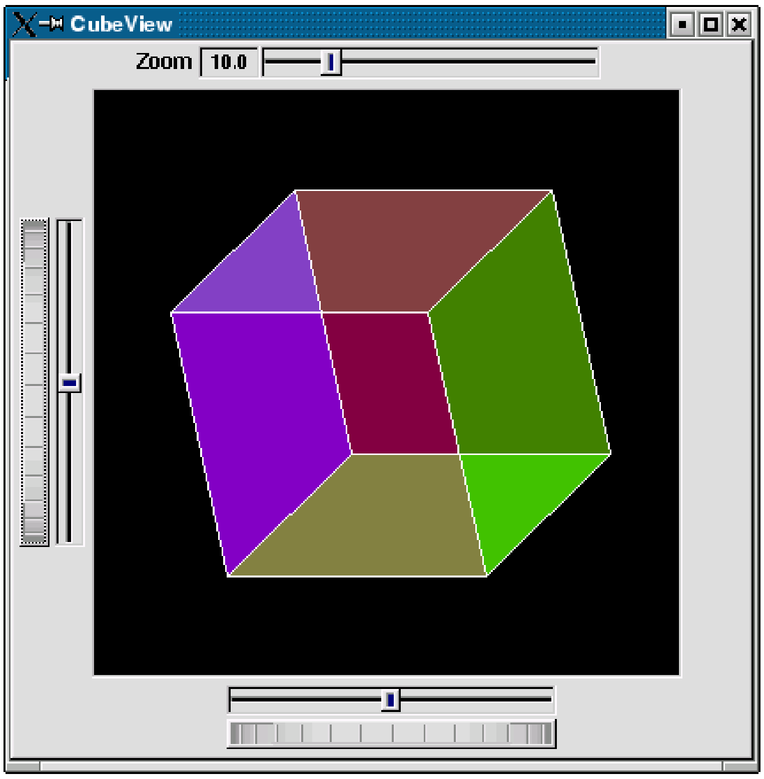


Figure 11.2: CubeView demo

The window is of class CubeViewUI, and is completely generated by FLUID, including class member functions. The central display of the cube is a separate subclass of Fl_Gl_Window called CubeView. CubeViewUI manages CubeView using callbacks from the various sliders and rollers to manipulate the viewing angle and zoom of CubeView.

At the completion of this tutorial you will (hopefully) understand how to:

1. Use FLUID to create a complete user interface class, including constructor and any member functions necessary.
2. Use FLUID to set callbacks member functions of a custom widget classes.
3. Subclass an Fl_Gl_Window to suit your purposes.

11.5.1 The CubeView Class

The CubeView class is a subclass of Fl_Gl_Window. It has methods for setting the zoom, the *x* and *y* pan, and the rotation angle about the *x* and *y* axes.

You can safely skip this section as long as you realize that CubeView is a sublass of Fl_Gl_Window and will respond to calls from CubeViewUI, generated by FLUID.

The CubeView Class Definition

Here is the CubeView class definition, as given by its header file ”test/CubeView.h”:

```
class CubeView : public Fl_Gl_Window {
  public:
    CubeView(int x,int y,int w,int h,const char *l=0);
    // this value determines the scaling factor used to draw the cube.
    double size;
    /* Set the rotation about the vertical (y ) axis.

     * This function is called by the horizontal roller in CubeViewUI
     * and the initialize button in CubeViewUI.
     */
    void v_angle(float angle){vAng=angle;};
    // Return the rotation about the vertical (y ) axis.
    float v_angle(){return vAng;};
    /* Set the rotation about the horizontal (x ) axis.

     * This function is called by the vertical roller in CubeViewUI
       and the
     * initialize button in CubeViewUI.
     */
    void h_angle(float angle){hAng=angle;};
    // the rotation about the horizontal (x ) axis.
    float h_angle(){return hAng;};
    /* Sets the x shift of the cube view camera.

     * This function is called by the slider in CubeViewUI and the
     * initialize button in CubeViewUI.
     */
    void panx(float x){xshift=x;};
    /* Sets the y shift of the cube view camera.

     * This function is called by the slider in CubeViewUI and the
     * initialize button in CubeViewUI.
     */
    void pany(float y){yshift=y;};
    /* The widget class draw() override.
     * The draw() function initialize Gl for another round of
     * drawing then calls specialized functions for drawing each
     * of the entities displayed in the cube view.
     */
    void draw();

  private:
    /* Draw the cube boundaries
     * Draw the faces of the cube using the boxv[] vertices, using
     * GL_LINE_LOOP for the faces. The color is #defined by
     * CUBECOLOR.
     */
    void drawCube();

    float vAng,hAng; float xshift,yshift;

    float boxv0[3];float boxv1[3]; float boxv2[3];float boxv3[3];
    float boxv4[3];float boxv5[3]; float boxv6[3];float boxv7[3];
};
```

The CubeView Class Implementation

Here is the CubeView implementation. It is very similar to the ”cube” demo included with FLTK.

```
#include "CubeView.h"
#include <math.h>

CubeView::CubeView(int x,int y,int w,int h,const char *l)
            : Fl_Gl_Window(x,y,w,h,l)
{
    vAng = 0.0; hAng=0.0; size=10.0;
    /* The cube definition. These are the vertices of a unit cube
     * centered on the origin.*/
    boxv0[0] = -0.5; boxv0[1] = -0.5; boxv0[2] = -0.5; boxv1[0] = 0.5;
    boxv1[1] = -0.5; boxv1[2] = -0.5; boxv2[0] = 0.5; boxv2[1] = 0.5;
    boxv2[2] = -0.5; boxv3[0] = -0.5; boxv3[1] = 0.5; boxv3[2] = -0.5;
    boxv4[0] = -0.5; boxv4[1] = -0.5; boxv4[2] = 0.5; boxv5[0] = 0.5;
    boxv5[1] = -0.5; boxv5[2] = 0.5; boxv6[0] = 0.5; boxv6[1] = 0.5;
    boxv6[2] = 0.5; boxv7[0] = -0.5; boxv7[1] = 0.5; boxv7[2] = 0.5;
};

// The color used for the edges of the bounding cube.
#define CUBECOLOR 255,255,255,255

void CubeView::drawCube() {
```

```
/* Draw a colored cube */
#define ALPHA 0.5
    glShadeModel(GL_FLAT);

    glBegin(GL_QUADS);
      glColor4f(0.0, 0.0, 1.0, ALPHA);
      glVertex3fv(boxv0);
      glVertex3fv(boxv1);
      glVertex3fv(boxv2);
      glVertex3fv(boxv3);

      glColor4f(1.0, 1.0, 0.0, ALPHA);
      glVertex3fv(boxv0);
      glVertex3fv(boxv4);
      glVertex3fv(boxv5);
      glVertex3fv(boxv1);

      glColor4f(0.0, 1.0, 1.0, ALPHA);
      glVertex3fv(boxv2);
      glVertex3fv(boxv6);
      glVertex3fv(boxv7);
      glVertex3fv(boxv3);

      glColor4f(1.0, 0.0, 0.0, ALPHA);
      glVertex3fv(boxv4);
      glVertex3fv(boxv5);
      glVertex3fv(boxv6);
      glVertex3fv(boxv7);

      glColor4f(1.0, 0.0, 1.0, ALPHA);
      glVertex3fv(boxv0);
      glVertex3fv(boxv3);
      glVertex3fv(boxv7);
      glVertex3fv(boxv4);

      glColor4f(0.0, 1.0, 0.0, ALPHA);
      glVertex3fv(boxv1);
      glVertex3fv(boxv5);
      glVertex3fv(boxv6);
      glVertex3fv(boxv2);
    glEnd();

    glColor3f(1.0, 1.0, 1.0);
    glBegin(GL_LINES);
      glVertex3fv(boxv0);
      glVertex3fv(boxv1);

      glVertex3fv(boxv1);
      glVertex3fv(boxv2);

      glVertex3fv(boxv2);
      glVertex3fv(boxv3);

      glVertex3fv(boxv3);
      glVertex3fv(boxv0);

      glVertex3fv(boxv4);
      glVertex3fv(boxv5);

      glVertex3fv(boxv5);
      glVertex3fv(boxv6);

      glVertex3fv(boxv6);
      glVertex3fv(boxv7);

      glVertex3fv(boxv7);
      glVertex3fv(boxv4);

      glVertex3fv(boxv0);
      glVertex3fv(boxv4);

      glVertex3fv(boxv1);
      glVertex3fv(boxv5);

      glVertex3fv(boxv2);
      glVertex3fv(boxv6);

      glVertex3fv(boxv3);
      glVertex3fv(boxv7);
    glEnd();
```

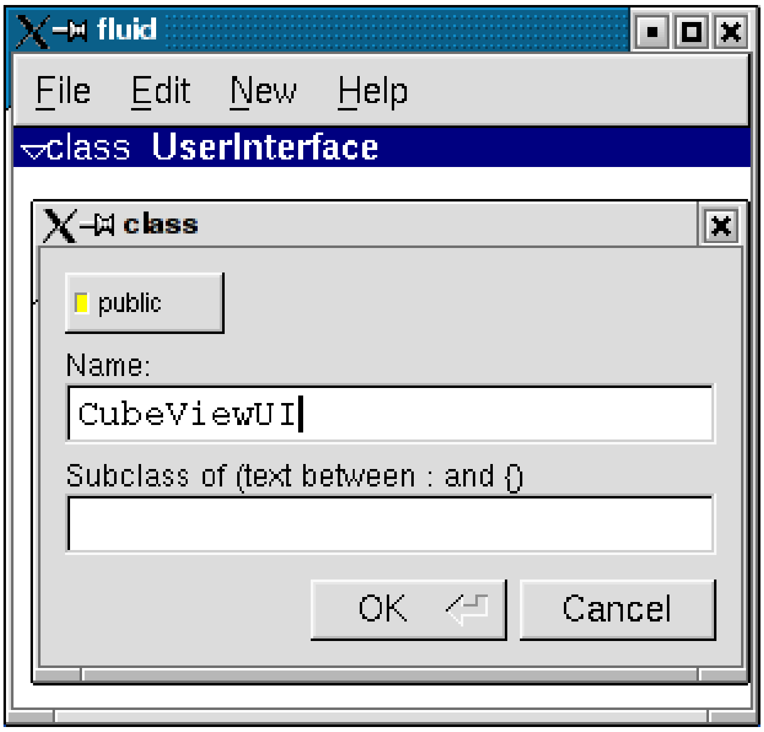
```
};//drawCube

void CubeView::draw() {
    if (!valid()) {
        glLoadIdentity(); glViewport(0,0,w(),h());
        glOrtho(-10,10,-10,10,-20000,10000); glEnable(GL_BLEND);
        glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    }

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPushMatrix(); glTranslatef(xshift, yshift, 0);
    glRotatef(hAng,0,1,0); glRotatef(vAng,1,0,0);
    glScalef(float(size),float(size),float(size)); drawCube();
    glPopMatrix();
};
```

11.5.2 The CubeViewUI Class

We will completely construct a window to display and control the CubeView defined in the previous section using FLUID.

Defining the CubeViewUI Class

Once you have started FLUID, the first step in defining a class is to create a new class within FLUID using the **New->Code->Class** menu item. Name the class "CubeViewUI" and leave the subclass blank. We do not need any inheritance for this window. You should see the new class declaration in the FLUID browser window.



class UserInterface

Name:
CubeViewUI

Subclass of (text between : and {)

Figure 11.3: FLUID file for CubeView

Adding the Class Constructor

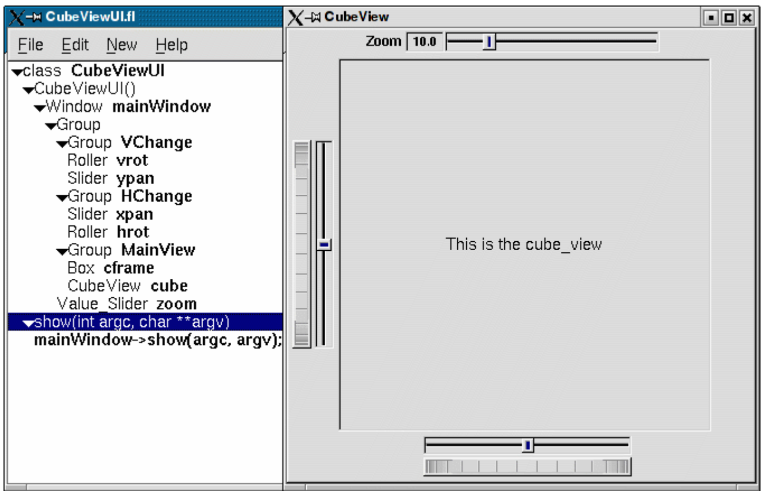
Click on the CubeViewUI class in the FLUID window and add a new method by selecting **New->Code->Function/Method.** The name of the function will also be CubeViewUI. FLUID will understand that this will be the constructor for the class and will generate the appropriate code. Make sure you declare the constructor public.

Then add a window to the CubeViewUI class. Highlight the name of the constructor in the FLUID browser window and click on **New->Group->Window**. In a similar manner add the following to the CubeViewUI constructor:

- A horizontal roller named `hrot`
- A vertical roller named `vrot`
- A horizontal slider named `xpan`
- A vertical slider named `ypan`
- A horizontal value slider named `zoom`

None of these additions need be public. And they shouldn't be unless you plan to expose them as part of the interface for CubeViewUI.

When you are finished you should have something like this:



```
class CubeViewUI
  CubeViewUI()
    Window mainWindow
      Group
        Group VChange
          Roller vrot
          Slider ypan
        Group HChange
          Slider xpan
          Roller hrot
        Group MainView
          Box cframe
          CubeView cube
        Value_Slider zoom
  show(int argc, char **argv)
    mainWindow->show(argc, argv);
```

Figure 11.4: FLUID window containing CubeView demo

We will talk about the `show()` method that is highlighted shortly.

Adding the CubeView Widget

What we have is nice, but does little to show our cube. We have already defined the CubeView class and we would like to show it within the CubeViewUI.

The CubeView class inherits the Fl_Gl_Window class, which is created in the same way as a Fl_Box widget. Use **New->Other->Box** to add a square box to the main window. This will be no ordinary box, however.

The Box properties window will appear. The key to letting CubeViewUI display CubeView is to enter CubeView in the **Class:** text entry box. This tells FLUID that it is not an Fl_Box, but a similar widget with the same constructor.

In the **Extra Code:** field enter `#include "CubeView.h"`

This `#include` is important, as we have just included CubeView as a member of CubeViewUI, so any public CubeView methods are now available to CubeViewUI.

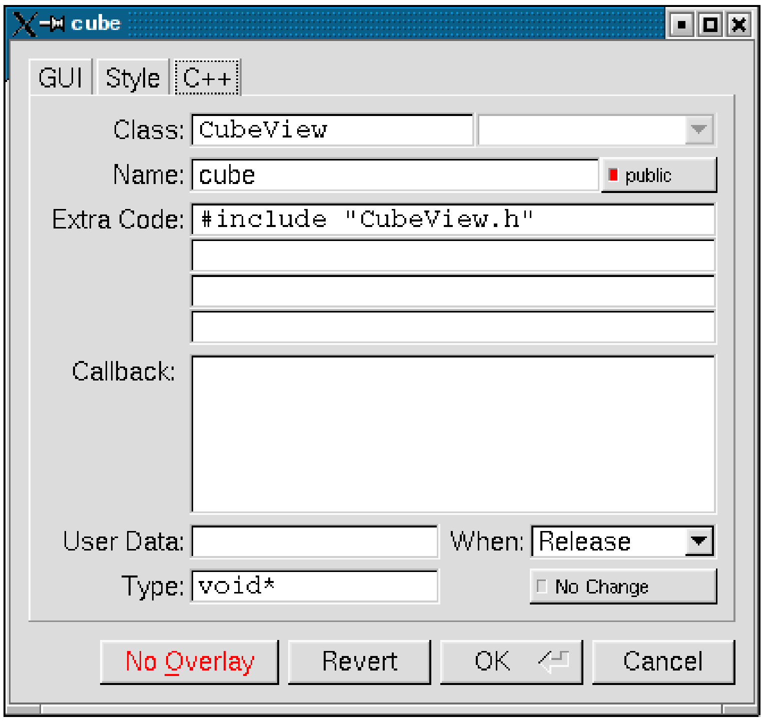

Class: CubeView
Name: cube
Extra Code: #include "CubeView.h"
Callback:
User Data:
When: Release
Type: void*

Figure 11.5: CubeView methods

Defining the Callbacks

Each of the widgets we defined before adding CubeView can have callbacks that call CubeView methods. You can call an external function or put in a short amount of code in the **Callback** field of the widget panel. For example, the callback for the `ypan` slider is:

```
cube->pany(((Fl_Slider *)o)->value());
cube->redraw();
```

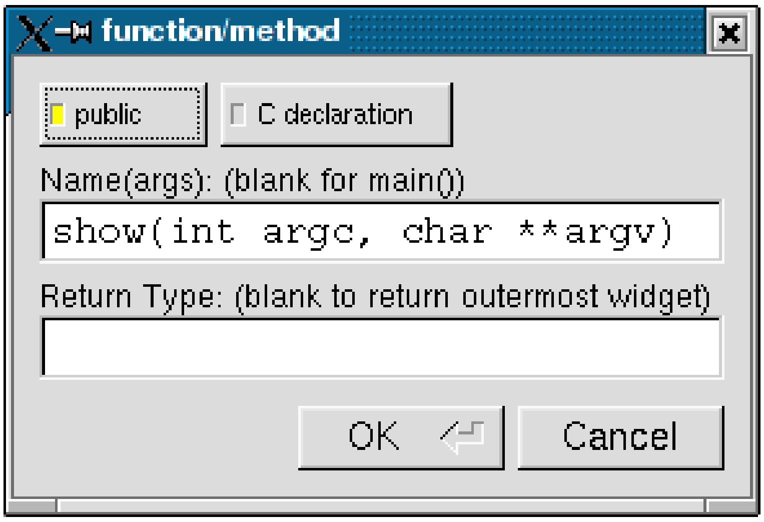
We call `cube->redraw()` after changing the value to update the CubeView window. CubeView could easily be modified to do this, but it is nice to keep this exposed. In the case where you may want to do more than one view change only redrawing once saves a lot of time.

There is no reason to wait until after you have added CubeView to enter these callbacks. FLUID assumes you are smart enough not to refer to members or functions that don't exist.

Adding a Class Method

You can add class methods within FLUID that have nothing to do with the GUI. As an example add a show function so that CubeViewUI can actually appear on the screen.

Make sure the top level CubeViewUI is selected and select **New->Code->Function/Method**. Just use the name `show()`. We don't need a return value here, and since we will not be adding any widgets to this method FLUID will assign it a return type of `void`.



Name(args): (blank for main())

```
show(int argc, char **argv)
```

Return Type: (blank to return outermost widget)

Figure 11.6: CubeView constructor

Once the new method has been added, highlight its name and select **New->Code->Code.** Enter the method's code in the code window.

11.5.3 Adding Constructor Initialization Code

If you need to add code to initialize a class, for example setting initial values of the horizontal and vertical angles in the CubeView, you can simply highlight the constructor and select **New->Code->Code**. Add any required code.

11.5.4 Generating the Code

Now that we have completely defined the CubeViewUI, we have to generate the code. There is one last trick to ensure this all works. Open the preferences dialog from **Edit->Preferences**.

At the bottom of the preferences dialog box is the key: **"Include Header from Code"**. Select that option and set your desired file extensions and you are in business. You can include the CubeViewUI.h (or whatever extension you prefer) as you would any other C++ class.

11.6 FLUID Reference

The following sections describe each of the windows in FLUID.

11.6.1 The Widget Browser

The main window shows a menu bar and a scrolling browser of all the defined widgets. The name of the `.fl` file being edited is shown in the window title.

The widgets are stored in a hierarchy. You can open and close a level by clicking the "triangle" at the left of a widget. The leftmost widgets are the *parents*, and all the widgets listed below them are their *children*. Parents don't have to have any children.

The top level of the hierarchy is composed of *functions* and *classes*. Each of these will produce a single C++ public function or class in the output `.cxx` file. Calling the function or instantiating the class will create all of the child widgets.

The second level of the hierarchy contains the *windows*. Each of these produces an instance of class Fl_Window.

Below that are either *widgets* (subclasses of Fl_Widget) or *groups* of widgets (including other groups). Plain groups are for layout, navigation, and resize purposes. *Tab groups* provide the well-known file-card tab interface.

Widgets are shown in the browser by either their *name* (such as "main_panel" in the example), or by their *type* and *label* (such as "Button "the green"").

You *select* widgets by clicking on their names, which highlights them (you can also select widgets from any displayed window). You can select many widgets by dragging the mouse across them, or by using Shift+Click to toggle them on and off. To select no widgets, click in the blank area under the last widget. Note that hidden children may be selected even when there is no visual indication of this.

You *open* widgets by double-clicking on them, or (to open several widgets you have picked) by typing the F1 key. A control panel will appear so you can change the widget(s).

11.6.2 Menu Items

The menu bar at the top is duplicated as a pop-up menu on any displayed window. The shortcuts for all the menu items work in any window. The menu items are:

File/Open... (Ctrl+o)

Discards the current editing session and reads in a different `.fl` file. You are asked for confirmation if you have changed the current file.

FLUID can also read `.fd` files produced by the Forms and XForms "fdesign" programs. It is best to File/Merge them instead of opening them. FLUID does not understand everything in a `.fd` file, and will print a warning message on the controlling terminal for all data it does not understand. You will probably need to edit the resulting setup to fix these errors. Be careful not to save the file without changing the name, as FLUID will write over the `.fd` file with its own format, which fdesign cannot read!

File/Insert... (Ctrl+i)

Inserts the contents of another `.fl` file, without changing the name of the current `.fl` file. All the functions (even if they have the same names as the current ones) are added, and you will have to use cut/paste to put the widgets where you want.

File/Save (Ctrl+s)

Writes the current data to the `.fl` file. If the file is unnamed then FLUID will ask for a filename.

File/Save As... (Ctrl+Shift+S)

Asks for a new filename and saves the file.

File/Write Code (Ctrl+Shift+C)

"Compiles" the data into a `.cxx` and `.h` file. These are exactly the same as the files you get when you run FLUID with the `-c` switch.

The output file names are the same as the `.fl` file, with the leading directory and trailing ".fl" stripped, and ".h" or ".cxx" appended.

File/Write Strings (Ctrl+Shift+W)

Writes a message file for all of the text labels defined in the current file.

The output file name is the same as the `.fl` file, with the leading directory and trailing ".fl" stripped, and ".txt", ".po", or ".msg" appended depending on the Internationalization Mode.

File/Quit (Ctrl+q)

Exits FLUID. You are asked for confirmation if you have changed the current file.

Edit/Undo (Ctrl+z)

This isn't implemented yet. You should do save often so you can recover from any mistakes you make.

Edit/Cut (Ctrl+x)

Deletes the selected widgets and all of their children. These are saved to a "clipboard" file and can be pasted back into any FLUID window.

Edit/Copy (Ctrl+c)

Copies the selected widgets and all of their children to the "clipboard" file.

Edit/Paste (Ctrl+c)

Pastes the widgets from the clipboard file.

If the widget is a window, it is added to whatever function is selected, or contained in the current selection.

If the widget is a normal widget, it is added to whatever window or group is selected. If none is, it is added to the window or group that is the parent of the current selection.

To avoid confusion, it is best to select exactly one widget before doing a paste.

Cut/paste is the only way to change the parent of a widget.

Edit/Select All (Ctrl+a)

Selects all widgets in the same group as the current selection.

If they are all selected already then this selects all widgets in that group's parent. Repeatedly typing Ctrl+a will select larger and larger groups of widgets until everything is selected.

Edit/Open... (F1 or double click)

Displays the current widget in the attributes panel. If the widget is a window and it is not visible then the window is shown instead.

Edit/Sort

Sorts the selected widgets into left to right, top to bottom order. You need to do this to make navigation keys in FLTK work correctly. You may then fine-tune the sorting with "Earlier" and "Later". This does not affect the positions of windows or functions.

Edit/Earlier (F2)

Moves all of the selected widgets one earlier in order among the children of their parent (if possible). This will affect navigation order, and if the widgets overlap it will affect how they draw, as the later widget is drawn on top of the earlier one. You can also use this to reorder functions, classes, and windows within functions.

Edit/Later (F3)

Moves all of the selected widgets one later in order among the children of their parent (if possible).

Edit/Group (F7)

Creates a new Fl_Group and make all the currently selected widgets children of it.

Edit/Ungroup (F8)

Deletes the parent group if all the children of a group are selected.

Edit/Overlays on/off (Ctrl+Shift+O)

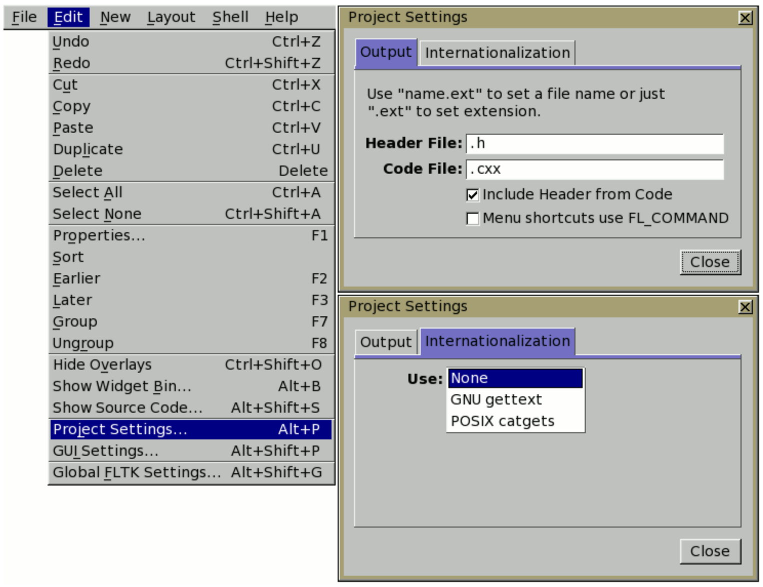
Toggles the display of the red overlays off, without changing the selection. This makes it easier to see box borders and how the layout looks. The overlays will be forced back on if you change the selection.

Edit/Project Settings... (Alt+p)

Displays the project settings panel.

Under the "Output" tab you control the extensions or names of the files that are generated by FLUID. If you check the "Include Header from Code" button the code file will include the header file automatically.

Under the "Internationalization" tab are the internationalization options, described later in this chapter.



Use "name.ext" to set a file name or just ".ext" to set extension.

Header File: .h

Code File: .cxx

☑ Include Header from Code

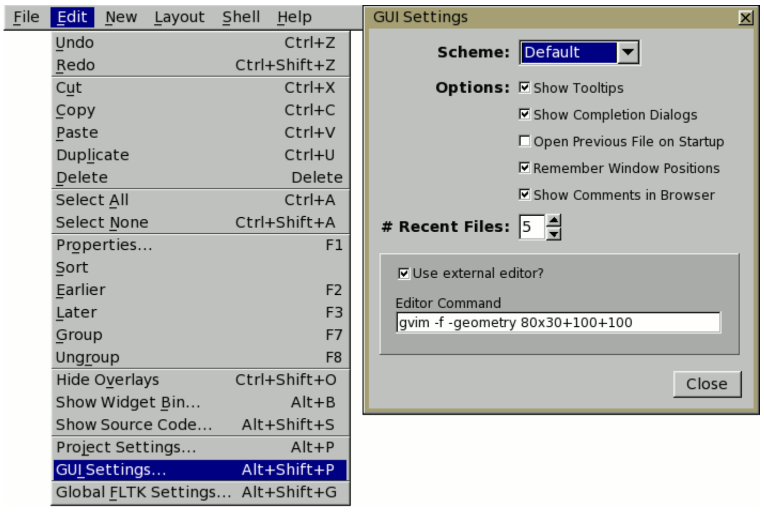
☐ Menu shortcuts use FL_COMMAND

Use: [✓ None] GNU gettext POSIX catgets

Figure 11.7: FLUID Project Settings Window

Edit/GUI Settings... (Shift+Alt+p)

Displays the GUI Settings panel, used to control the user interface settings.



Scheme: Default

Options: ☑ Show Tooltips
☑ Show Completion Dialogs
☐ Open Previous File on Startup
☑ Remember Window Positions
☑ Show Comments in Browser

Recent Files: 5

☑ Use external editor?

Editor Command: gvim -f -geometry 80x30+100+100

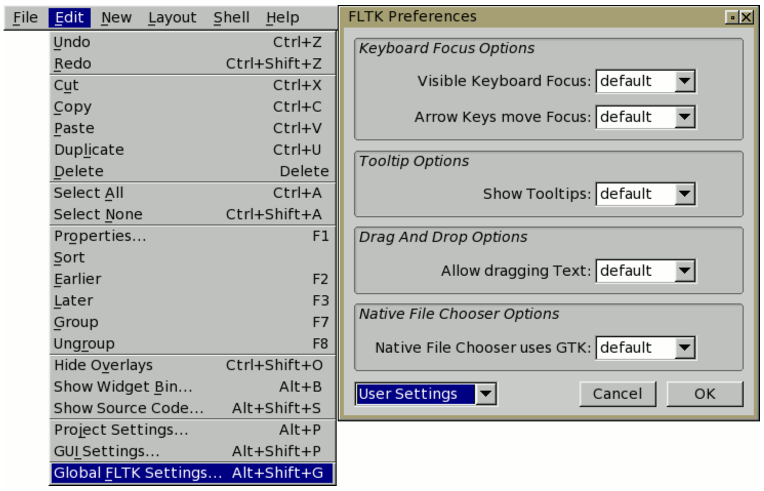
Figure 11.8: FLUID GUI Settings Window

Edit/Global FLTK Settings... (Shift+Alt+g)

Displays the FLTK Global Settings (”Preferences”) panel, used to control fluid’s user specific and/or system wide settings.

Tooltips provide descriptions of each option.

At the lower-right, ”User Settings” causes changes to only affect the current user, ”System Settings” causes changes to be applied to all users on the current machine.



Keyboard Focus Options
Visible Keyboard Focus: default
Arrow Keys move Focus: default

Tooltip Options
Show Tooltips: default

Drag And Drop Options
Allow dragging Text: default

Native File Chooser Options
Native File Chooser uses GTK: default

User Settings

Figure 11.9: FLUID Global Settings Window

New/Code/Function

Creates a new C function. You will be asked for a name for the function. This name should be a legal C++ function template, without the return type. You can pass arguments which can be referred to by code you type into the individual widgets.

If the function contains any unnamed windows, it will be declared as returning a Fl_Window pointer. The unnamed window will be returned from it (more than one unnamed window is useless). If the function contains only named windows, it will be declared as returning nothing (`void`).

It is possible to make the `.cxx` output be a self-contained program that can be compiled and executed. This is done by deleting the function name so `main(argc,argv)` is used. The function will call `show()` on all the windows it creates and then call `Fl::run()`. This can also be used to test resize behavior or other parts of the user interface.

You can change the function name by double-clicking on the function.

New/Window

Creates a new Fl_Window widget. The window is added to the currently selected function, or to the function containing the currently selected item. The window will appear, sized to 100x100. You can resize it to whatever size you require.

The widget panel will also appear and is described later in this chapter.

New/...

All other items on the New menu are subclasses of Fl_Widget. Creating them will add them to the currently selected group or window, or the group or window containing the currently selected widget. The initial dimensions and position are chosen by copying the current widget, if possible.

When you create the widget you will get the widget's control panel, which is described later in this chapter.

Layout/Align/...

Align all selected widgets to the first widget in the selection.

Layout/Space Evenly/...

Space all selected widgets evenly inside the selected space. Widgets will be sorted from first to last.

Layout/Make Same Size/...

Make all selected widgets the same size as the first selected widget.

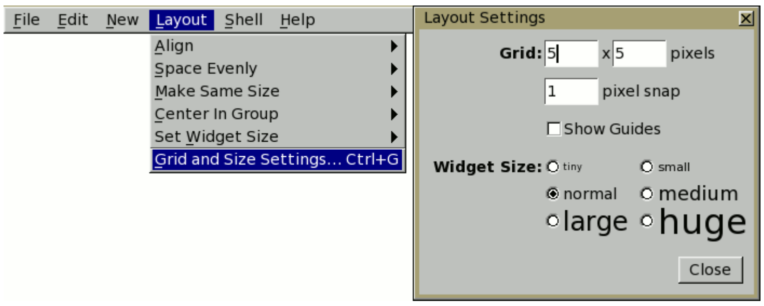
Layout/Center in Group/...

Center all selected widgets relative to their parent widget

Layout/Grid and Size Settings... (Ctrl+g)

Displays the grid settings panel.

This panel controls the grid that all widgets snap to when you move and resize them, and for the "snap" which is how far a widget has to be dragged from its original position to actually change.



Grid: 5 x 5 pixels
1 pixel snap
Show Guides
Widget Size: tiny small [✓ normal] medium large huge

Figure 11.10: FLUID Layout/Grid Settings Window

Shell/Execute Command... (Alt+x)

Displays the shell command panel. The shell command is commonly used to run a 'make' script to compile the FLTK output.

Shell/Execute Again (Alt+g)

Run the shell command again.

Help/About FLUID

Pops up a panel showing the version of FLUID.

Help/On FLUID

Shows this chapter of the manual.

Help/Manual

Shows the contents page of the manual

11.6.3 The Widget Panel

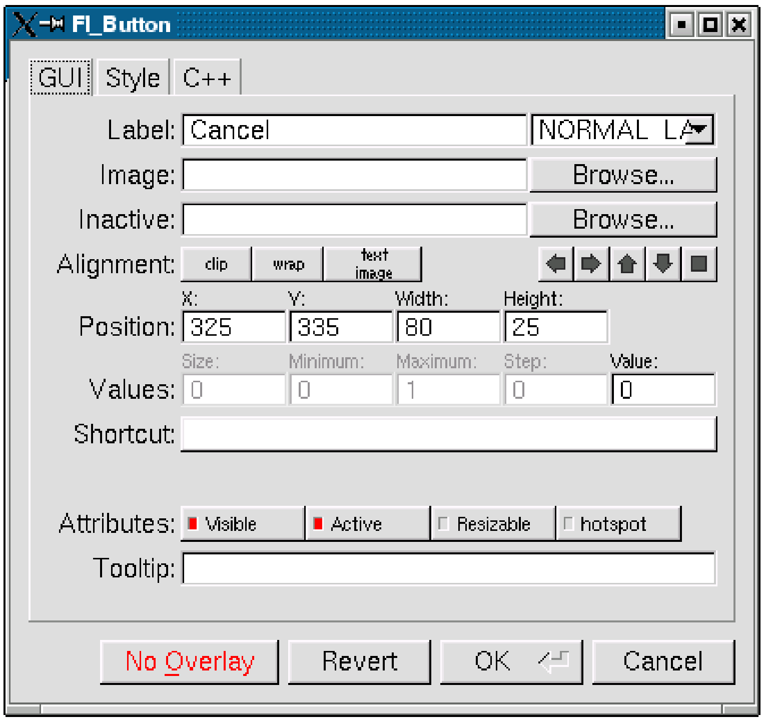
When you double-click on a widget or a set of widgets you will get the "widget attribute panel".

When you change attributes using this panel, the changes are reflected immediately in the window. It is useful to hit the "no overlay" button (or type Ctrl+Shift+O) to hide the red overlay so you can see the widgets more accurately, especially when setting the box type.

If you have several widgets selected, they may have different values for the fields. In this case the value for *one* of the widgets is shown. But if you change this value, *all* of the selected widgets are changed to the new value.

Hitting "OK" makes the changes permanent. Selecting a different widget also makes the changes permanent. FLUID checks for simple syntax errors such as mismatched parenthesis in any code before saving any text.

"Revert" or "Cancel" put everything back to when you last brought up the panel or hit OK. However in the current version of FLUID, changes to "visible" attributes (such as the color, label, box) are not undone by revert or cancel. Changes to code like the callbacks are undone, however.



Label: Cancel   NORMAL_LA
Image:
Inactive:
Alignment:
Position: X: 325 Y: 335 Width: 80 Height: 25
Values: Size: 0 Minimum: 0 Maximum: 1 Step: 0 Value: 0
Shortcut:
Attributes: ■ Visible ■ Active □ Resizable □ hotspot
Tooltip:

Figure 11.11: The FLUID widget GUI attributes

11.7 GUI Attributes

Label (text field)

String to print next to or inside the button. You can put newlines into the string to make multiple lines. The easiest way is by typing Ctrl+j.

Symbols can be added to the label using the at sign ("@").

Label (pull down menu)

How to draw the label. Normal, shadowed, engraved, and embossed change the appearance of the text.

Image

The active image for the widget. Click on the **Browse**... button to pick an image file using the file chooser.

Inactive

The inactive image for the widget. Click on the **Browse**... button to pick an image file using the file chooser.

Alignment (buttons)

Where to draw the label. The arrows put it on that side of the widget, you can combine them to put it in the corner. The ”box” button puts the label inside the widget, rather than outside.

The **clip** button clips the label to the widget box, the **wrap** button wraps any text in the label, and the **text image** button puts the text over the image instead of under the image.

Position (text fields)

The position fields show the current position and size of the widget box. Enter new values to move and/or resize a widget.

Values (text fields)

The values and limits of the current widget. Depending on the type of widget, some or all of these fields may be inactive.

Shortcut

The shortcut key to activate the widget. Click on the shortcut button and press any key sequence to set the shortcut.

Attributes (buttons)

The **Visible** button controls whether the widget is visible (on) or hidden (off) initially. Don’t change this for windows or for the immediate children of a Tabs group.

The **Active** button controls whether the widget is activated (on) or deactivated (off) initially. Most widgets appear greyed out when deactivated.

The **Resizable** button controls whether the window is resizeable. In addition all the size changes of a window or group will go ”into” the resizable child. If you have a large data display surrounded by buttons, you probably want that data area to be resizable. You can get more complex behavior by making invisible boxes the resizable widget, or by using hierarchies of groups. Unfortunately the only way to test it is to compile the program. Resizing the FLUID window is *not* the same as what will happen in the user program.

The **Hotspot** button causes the parent window to be positioned with that widget centered on the mouse. This position is determined *when the FLUID function is called*, so you should call it immediately before showing the window. If you want the window to hide and then reappear at a new position, you should have your program set the hotspot itself just before `show()`.

The **Border** button turns the window manager border on or off. On most window managers you will have to close the window and reopen it to see the effect.

X Class (text field)

The string typed into here is passed to the X window manager as the class. This can change the icon or window decorations. On most (all?) window managers you will have to close the window and reopen it to see the effect.

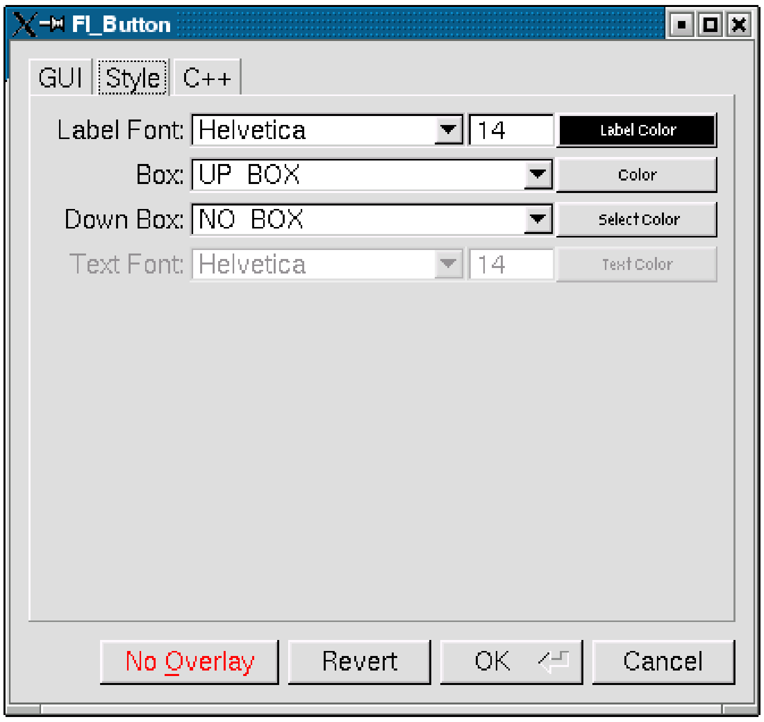


Figure 11.12: The FLUID widget Style attributes

11.7.1 Style Attributes

Label Font (pulldown menu)

Font to draw the label in. Ignored by symbols, bitmaps, and pixmaps. Your program can change the actual font used by these ”slots” in case you want some font other than the 16 provided.

Label Size (pulldown menu)

Pixel size (height) for the font to draw the label in. Ignored by symbols, bitmaps, and pixmaps. To see the result without dismissing the panel, type the new number and then Tab.

Label Color (button)

Color to draw the label. Ignored by pixmaps (bitmaps, however, do use this color as the foreground color).

Box (pulldown menu)

The boxtype to draw as a background for the widget.

Many widgets will work, and draw faster, with a ”frame” instead of a ”box”. A frame does not draw the colored interior, leaving whatever was already there visible. Be careful, as FLUID may draw this ok but the real program may leave unwanted stuff inside the widget.

If a window is filled with child widgets, you can speed up redrawing by changing the window’s box type to ”NO_BOX”. FLUID will display a checkerboard for any areas that are not colored in by boxes. Note that this checkerboard is not drawn by the resulting program. Instead random garbage will be displayed.

Down Box (pulldown menu)

The boxtype to draw when a button is pressed or for some parts of other widgets like scrollbars and valuators.

Color (button)

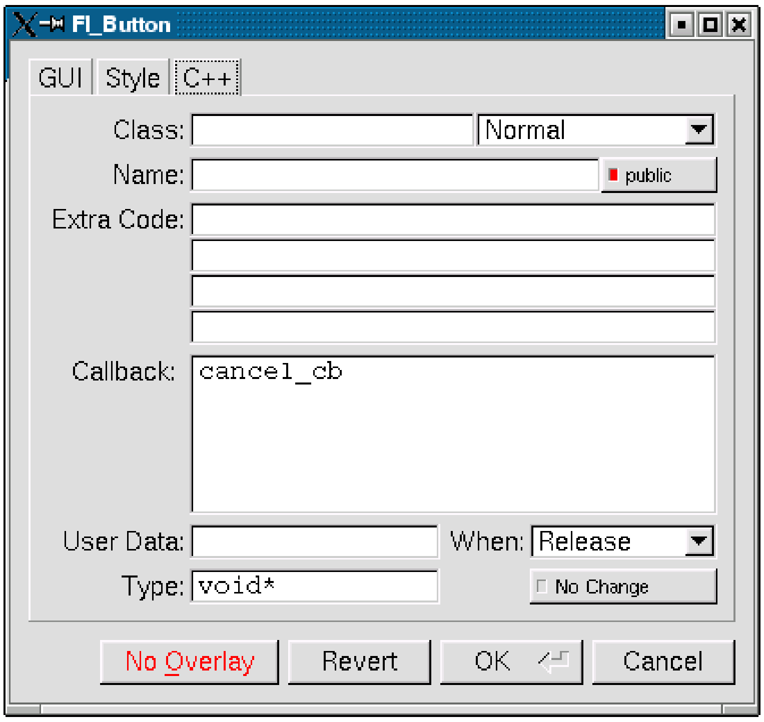
The color to draw the box with.

Select Color (button)

Some widgets will use this color for certain parts. FLUID does not always show the result of this: this is the color buttons draw in when pushed down, and the color of input fields when they have the focus.

Text Font, Size, and Color

Some widgets display text, such as input fields, pull-down menus, and browsers.



Class: | Normal
Name:
Extra Code:
Callback: cancel_cb
User Data:
When: Release
Type: void*
No Change

Figure 11.13: The FLUID widget C++ attributes

11.7.2 C++ Attributes

Class

This is how you use your own subclasses of Fl_Widget. Whatever identifier you type in here will be the class that is instantiated.

In addition, no `#include` header file is put in the `.h` file. You must provide a `#include` line as the first line of the "Extra Code" which declares your subclass.

The class must be similar to the class you are spoofing. It does not have to be a subclass. It is sometimes useful to change this to another FLTK class. Currently the only way to get a double-buffered window is to change this field for the window to "Fl_Double_Window" and to add

```
#include <FL/Fl_Double_Window.h>
```

to the extra code.

Type (upper-right pulldown menu)

Some classes have subtypes that modify their appearance or behavior. You pick the subtype off of this menu.

Name (text field)

Name of a variable to declare, and to store a pointer to this widget into. This variable will be of type "<class>*". If the name is blank then no variable is created.

You can name several widgets with "name[0]", "name[1]", "name[2]", etc. This will cause FLUID to declare an array of pointers. The array is big enough that the highest number found can be stored. All widgets in the array must be the same type.

Public (button)

Controls whether the widget is publicly accessible. When embedding widgets in a C++ class, this controls whether the widget is `public` or `private` in the class. Otherwise it controls whether the widget is declared `static` or global (`extern`).

Extra Code (text fields)

These four fields let you type in literal lines of code to dump into the `.h` or `.cxx` files.

If the text starts with a `#` or the word `extern` then FLUID thinks this is an "include" line, and it is written to the `.h` file. If the same include line occurs several times then only one copy is written.

All other lines are "code" lines. The current widget is pointed to by the local variable `o`. The window being constructed is pointed to by the local variable `w`. You can also access any arguments passed to the function here, and any named widgets that are before this one.

FLUID will check for matching parenthesis, braces, and quotes, but does not do much other error checking. Be careful here, as it may be hard to figure out what widget is producing an error in the compiler. If you need more than four lines you probably should call a function in your own `.cxx` code.

Callback (text field)

This can either be the name of a function, or a small snippet of code. If you enter anything other than letters, numbers, and the underscore then FLUID treats it as code.

A name refers to a function in your own code. It must be declared as `void name(<class>*,void*)`.

A code snippet is inserted into a static function in the `.cxx` output file. The function prototype is `void name(class *o, void *v)` so that you can refer to the widget as `o` and the `user_-data()` as `v`. FLUID will check for matching parenthesis, braces, and quotes, but does not do much other error checking. Be careful here, as it may be hard to figure out what widget is producing an error in the compiler.

If the callback is blank then no callback is set.

User Data (text field)

This is a value for the `user_data()` of the widget. If blank the default value of zero is used. This can be any piece of C code that can be cast to a `void` pointer.

Type (text field)

The `void*` in the callback function prototypes is replaced with this. You may want to use `long` for old XForms code. Be warned that anything other than `void*` is not guaranteed to work! However on most architectures other pointer types are ok, and `long` is usually ok, too.

When (pulldown menu)

When to do the callback. This can be **Never**, **Changed**, **Release**, or **Enter** Key. The value of **Enter Key** is only useful for text input fields.

There are other rare but useful values for the `when()` field that are not in the menu. You should use the extra code fields to put these values in.

No Change (button)

The **No Change** button means the callback is done on the matching event even if the data is not changed.

11.8 Selecting and Moving Widgets

Double-clicking a window name in the browser will display it, if not displayed yet. From this display you can select widgets, sets of widgets, and move or resize them. To close a window either double-click it or type `ESC`.

To select a widget, click it. To select several widgets drag a rectangle around them. Holding down shift will toggle the selection of the widgets instead.

You cannot pick hidden widgets. You also cannot choose some widgets if they are completely overlapped by later widgets. Use the browser to select these widgets.

The selected widgets are shown with a red ”overlay” line around them. You can move the widgets by dragging this box. Or you can resize them by dragging the outer edges and corners. Hold down the Alt key while dragging the mouse to defeat the snap-to-grid effect for fine positioning.

If there is a tab box displayed you can change which child is visible by clicking on the file tabs. The child you pick is selected.

The arrow, tab, and shift+tab keys ”navigate” the selection. Left, right, tab, or shift+tab move to the next or previous widgets in the hierarchy. Hit the right arrow enough and you will select every widget in the window. Up/down widgets move to the previous/next widgets that overlap horizontally. If the navigation does not seem to work you probably need to ”Sort” the widgets. This is important if you have input fields, as FLTK uses the same rules when using arrow keys to move between input fields.

To ”open” a widget, double click it. To open several widgets select them and then type F1 or pick ”Edit/Open” off the pop-up menu.

Type Ctrl+o to temporarily toggle the overlay off without changing the selection, so you can see the widget borders.

You can resize the window by using the window manager border controls. FLTK will attempt to round the window size to the nearest multiple of the grid size and makes it big enough to contain all the widgets (it does this using illegal X methods, so it is possible it will barf with some window managers!). Notice that the actual window in your program may not be resizable, and if it is, the effect on child widgets may be different.

The panel for the window (which you get by double-clicking it) is almost identical to the panel for any other Fl_Widget. There are three extra items:

11.9 Image Labels

The *contents* of the image files in the **Image** and **Inactive** text fields are written to the `.cxx` file. If many widgets share the same image then only one copy is written. Since the image data is embedded in the generated source code, you need only distribute the C++ code and not the image files themselves.

However, the *filenames* are stored in the `.fl` file so you will need the image files as well to read the `.fl` file. Filenames are relative to the location of the `.fl` file and not necessarily the current directory. We recommend you either put the images in the same directory as the `.fl` file, or use absolute path names.

Notes for All Image Types

FLUID runs using the default visual of your X server. This may be 8 bits, which will give you dithered images. You may get better results in your actual program by adding the code ”Fl::visual(FL_RGB)” to your code right before the first window is displayed.

All widgets with the same image on them share the same code and source X pixmap. Thus once you have put an image on a widget, it is nearly free to put the same image on many other widgets.

If you edit an image at the same time you are using it in FLUID, the only way to convince FLUID to read the image file again is to remove the image from all widgets that are using it or re-load the `.fl` file.

Don’t rely on how FLTK crops images that are outside the widget, as this may change in future versions! The cropping of inside labels will probably be unchanged.

To more accurately place images, make a new ”box” widget and put the image in that as the label.

XBM (X Bitmap) Files

FLUID reads X bitmap files which use C source code to define a bitmap. Sometimes they are stored with the ”.h” or ”.bm” extension rather than the standard ”.xbm” extension.

FLUID writes code to construct an Fl_Bitmap image and use it to label the widget. The ’1’ bits in the bitmap are drawn using the label color of the widget. You can change this color in the FLUID widget attributes panel. The ’0’ bits are transparent.

The program ”bitmap” on the X distribution does an adequate job of editing bitmaps.

XPM (X Pixmap) Files

FLUID reads X pixmap files as used by the `libxpm` library. These files use C source code to define a pixmap. The filenames usually have the ”.xpm” extension.

FLUID writes code to construct an Fl_Pixmap image and use it to label the widget. The label color of the widget is ignored, even for 2-color images that could be a bitmap. XPM files can mark a single color as being transparent, and FLTK uses this information to generate a transparency mask for the image.

We have not found any good editors for small iconic pictures. For pixmaps we have used `XPaint` and the KDE icon editor.

BMP Files

FLUID reads Windows BMP image files which are often used in WIN32 applications for icons. F-LUID converts BMP files into (modified) XPM format and uses a Fl_BMP_Image image to label the widget. Transparency is handled the same as for XPM files. All image data is uncompressed when written to the source file, so the code may be much bigger than the `.bmp` file.

GIF Files

FLUID reads GIF image files which are often used in HTML documents to make icons. FLUID converts GIF files into (modified) XPM format and uses a Fl_GIF_Image image to label the widget. Transparency is handled the same as for XPM files. All image data is uncompressed when written to the source file, so the code may be much bigger than the `.gif` file. Only the first image of an animated GIF file is used.

JPEG Files

If FLTK is compiled with JPEG support, FLUID can read JPEG image files which are often used for digital photos. FLUID uses a Fl_JPEG_Image image to label the widget, and writes uncompressed RGB or grayscale data to the source file.

PNG (Portable Network Graphics) Files

If FLTK is compiled with PNG support, FLUID can read PNG image files which are often used in HTML documents. FLUID uses a Fl_PNG_Image image to label the widget, and writes uncompressed RGB or grayscale data to the source file. PNG images can provide a full alpha channel for partial transparency, and FLTK supports this as best as possible on each platform.

11.10 Internationalization with FLUID

FLUID supports internationalization (I18N for short) of label strings used by widgets. The preferences window (`Ctrl+p`) provides access to the I18N options.

11.10.1 I18N Methods

FLUID supports three methods of I18N: use none, use GNU gettext, and use POSIX catgets. The "use none" method is the default and just passes the label strings as-is to the widget constructors.

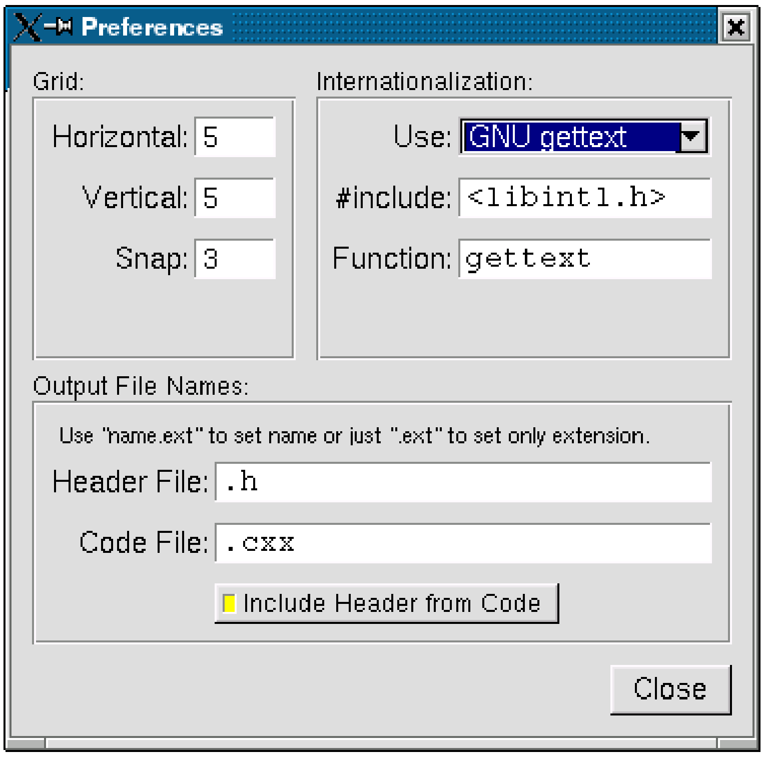
The "GNU gettext" method uses GNU gettext (or a similar text-based I18N library) to retrieve a localized string before calling the widget constructor.

The "POSIX catgets" method uses the POSIX catgets function to retrieve a numbered message from a message catalog before calling the widget constructor.

11.10.2 Using GNU gettext for I18N

FLUID's code support for GNU gettext is limited to calling a function or macro to retrieve the localized label; you still need to call `setlocale()` and `textdomain()` or `bindtextdomain()` to select the appropriate language and message file.

To use GNU gettext for I18N, open the preferences window and choose "GNU gettext" from the **Use:** chooser. Two new input fields will then appear to control the include file and function/macro name to use when retrieving the localized label strings.



Horizontal: 5
Vertical: 5
Snap: 3
Use: GNU gettext
#include: <libintl.h>
Function: gettext
Use "name.ext" to set name or just ".ext" to set only extension.
Header File: .h
Code File: .cxx

Figure 11.14: Internationalization using GNU gettext

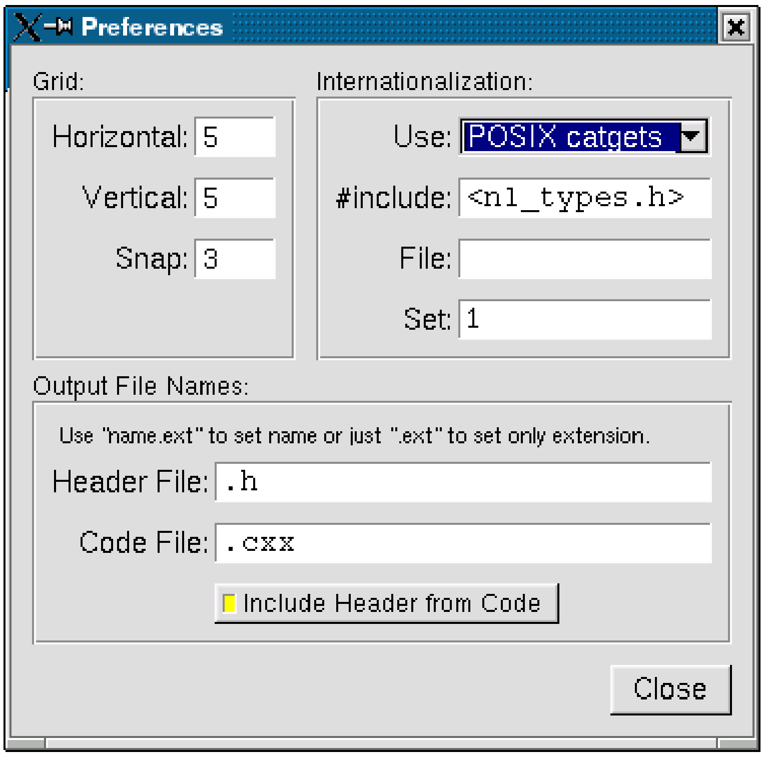
The **#include** field controls the header file to include for I18N; by default this is <**libintl.h**>, the standard I18N file for GNU gettext.

The **Function:** field controls the function (or macro) that will retrieve the localized message; by default the `gettext` function will be called.

11.10.3 Using POSIX catgets for I18N

FLUID's code support for POSIX catgets allows you to use a global message file for all interfaces or a file specific to each `.fl` file; you still need to call `setlocale()` to select the appropriate language.

To use POSIX catgets for I18N, open the preferences window and choose "POSIX catgets" from the **Use:** chooser. Three new input fields will then appear to control the include file, catalog file, and set number for retrieving the localized label strings.



Preferences

Grid:

Horizontal: 5

Vertical: 5

Snap: 3

Internationalization:

Use: POSIX catgets

#include: <nl_types.h>

File:

Set: 1

Output File Names:

Use "name.ext" to set name or just ".ext" to set only extension.

Header File: .h

Code File: .cxx

Include Header from Code

Close

Figure 11.15: Internationalization using POSIX catgets

The **#include** field controls the header file to include for I18N; by default this is <**nl_types.h**>, the standard I18N file for POSIX catgets.

The **File:** field controls the name of the catalog file variable to use when retrieving localized messages; by default the file field is empty which forces a local (static) catalog file to be used for all of the windows defined in your `.fl` file.

The **Set:** field controls the set number in the catalog file. The default set is 1 and rarely needs to be changed.

11.11 Known limitations

Declaration Blocks can be used to temporarily block out already designed code using `#if 0` and `#endif` type construction. This will effectively avoid compilation of blocks of code. However, static code and data generated by this segment (menu items, images, include statements, etc.) will still be generated and likely cause compile-time warnings.

Chapter 12

Advanced FLTK

This chapter explains advanced programming and design topics that will help you to get the most out of FLTK.

12.1 Multithreading

FLTK can be used to implement a GUI for a multithreaded application but, as with multithreaded programming generally, there are some concepts and caveats that must be kept in mind.

Key amongst these is that, for many of the target platforms on which FLTK is supported, only the `main()` thread of the process is permitted to handle system events, create or destroy windows and open or close windows. Further, only the `main()` thread of the process can safely write to the display.

To support this in a portable way, all FLTK `draw()` methods are executed in the `main()` thread. A worker thread may update the state of an existing widget, but it may not do any rendering directly, nor create or destroy a window. (**NOTE:** A special case exists for Fl_Gl_Window where it can, with suitable precautions, be possible to safely render to an existing GL context from a worker thread.)

Creating portable threads

We do not provide a threading interface as part of the library. A simple example showing how threads can be implemented, for all supported platforms, can be found in `test/threads.h` and `test/threads.cxx`.

FLTK has been used with a variety of thread interfaces, so if the simple example shown in `test/threads.cxx` does not cover your needs, you might want to select a third-party library that provides the features you require.

12.2 FLTK multithread locking - Fl::lock() and Fl::unlock()

In a multithreaded program, drawing of widgets (in the `main()` thread) happens asynchronously to widgets being updated by worker threads, so no drawing can occur safely whilst a widget is being modified (and no widget should be modified whilst drawing is in progress).

FLTK supports multithreaded applications using a locking mechanism internally. This allows a worker thread to lock the rendering context, preventing any drawing from taking place, whilst it changes the value of its widget.

Note

> The converse is also true; whilst a worker thread holds the lock, the `main()` thread may not be able to process any drawing requests, nor service any events. So a worker thread that holds the FLTK lock **must** contrive to do so for the shortest time possible or it could impair operation of the application.

The lock operates broadly as follows.

Using the FLTK library, the `main()` thread holds the lock whenever it is processing events or redrawing the display. It acquires (locks) and releases (unlocks) the FLTK lock automatically and no ”user intervention” is required. Indeed, a function that runs in the context of the `main()` thread ideally should **not** acquire / release the FLTK lock explicitly. (Though note that the lock calls are recursive, so calling Fl::lock() from a thread that already holds the lock, including the `main()` thread, is benign. The only constraint is that every call to Fl::lock() **must** be balanced by a corresponding call to Fl::unlock() to ensure the lock count is preserved.)

The `main()` thread **must** call Fl::lock() **once** before any windows are shown, to enable the internal lock (it is ”off” by default since it is not useful in single-threaded applications) but thereafter the `main()` thread lock is managed by the library internally.

A worker thread, when it wants to alter the value of a widget, can acquire the lock using Fl::lock(), update the widget, then release the lock using Fl::unlock(). Acquiring the lock ensures that the worker thread can update the widget, without any risk that the `main()` thread will attempt to redraw the widget whilst it is being updated.

Note that acquiring the lock is a blocking action; the worker thread will stall for as long as it takes to acquire the lock. If the `main()` thread is engaged in some complex drawing operation this may block the worker thread for a long time, effectively serializing what ought to be parallel operations. (This frequently comes as a surprise to coders less familiar with multithreaded programming issues; see the discussion of ”lockless programming” later for strategies for managing this.)

To incorporate the locking mechanism in the library, FLTK must be compiled with `--enable-threads` set during the `configure` process. IDE-based versions of FLTK are automatically compiled with the locking mechanism incorporated if possible. Since version 1.3, the `configure` script that builds the FLTK library also sets `--enable-threads` by default.

12.3 Simple multithreaded examples using Fl::lock

In `main()`, call Fl::lock() once before Fl::run() or Fl::wait() to enable the lock and start the runtime multithreading support for your program. All callbacks and derived functions like `handle()` and `draw()` will now be properly locked.

This might look something like this:

```
int main(int argc, char **argv) {
  /* Create your windows and widgets here */

  Fl::lock(); /* "start" the FLTK lock mechanism */

  /* show your window */
  main_win->show(argc, argv);

  /* start your worker threads */
  ... start threads ...

  /* Run the FLTK main loop */
  int result = Fl::run();

  /* terminate any pending worker threads */
  ... stop threads ...

  return result;
}
```

You can start as many threads as you like. From within a thread (other than the `main()` thread) FLTK calls must be wrapped with calls to Fl::lock() and Fl::unlock():

```
void my_thread(void) {
  while (thread_still_running) {
    /* do thread work */
    ...
    /* compute new values for widgets */
    ...

    Fl::lock();      // acquire the lock
    my_widget->update(values);
    Fl::unlock();     // release the lock; allow other threads to access FLTK again
    Fl::awake();      // use Fl::awake() to signal main thread to refresh the GUI
  }
}
```

Note

To trigger a refresh of the GUI from a worker thread, the worker code should call Fl::awake()

Using Fl::awake thread messages

You can send messages from worker threads to the `main()` thread using Fl::awake(void∗ message). If using this thread message interface, your `main()` might look like this:

```
int main(int argc, char **argv) {
  /* Create your windows and widgets here */

  Fl::lock(); /* "start" the FLTK lock mechanism */

  /* show your window */
  main_win->show(argc, argv);

  /* start your worker threads */
  ... start threads ...

  /* Run the FLTK loop and process thread messages */
  while (Fl::wait() > 0) {
    if ((next_message = Fl::thread_message()) != NULL) {
      /* process your data, update widgets, etc. */
      ...
    }
  }

  /* terminate any pending worker threads */
  ... stop threads ...

  return 0;
}
```

Your worker threads can send messages to the `main()` thread using Fl::awake(void∗ message):

```
void *msg;       // "msg" is a pointer to your message
Fl::awake(msg);  // send "msg" to main thread
```

A message can be anything you like. The `main()` thread can retrieve the message by calling Fl::thread_message().

Using Fl::awake callback messages

You can also request that the `main()` thread call a function on behalf of the worker thread by using Fl::awake(Fl_Awake_Handler cb, void∗ userdata).

The `main()` thread will execute the callback "as soon as possible" when next processing the pending events. This can be used by a worker thread to perform operations (for example showing or hiding windows) that are prohibited in a worker thread.

```
void do_something_cb(void *userdata) {
  // Will run in the context of the main thread
  ... do_stuff ...
}

// running in worker thread
void *data;                       // "data" is a pointer to your user data
Fl::awake(do_something_cb, data); // call to execute cb in main thread
```

Note

> The `main()` thread will execute the Fl_Awake_Handler callback `do_something_cb` asynchronously to the worker thread, at some short but indeterminate time after the worker thread registers the request. When it executes the Fl_Awake_Handler callback, the `main()` thread will use the contents of `*userdata` **at the time of execution**, not necessarily the contents that `*userdata` had at the time that the worker thread posted the callback request. The worker thread should therefore contrive **not** to alter the contents of `*userdata` once it posts the callback, since the worker thread does not know when the `main()` thread will consume that data. It is often useful that `userdata` point to a struct, one member of which the `main()` thread can modify to indicate that it has consumed the data, thereby allowing the worker thread to re-use or update `userdata`.

Warning

> The mechanisms used to deliver Fl::awake(void∗ message) and Fl::awake(Fl_Awake_Handler cb, void∗ userdata) events to the `main()` thread can interact in unexpected ways on some platforms. Therefore, for reliable operation, it is advised that a program use either Fl::awake(Fl_Awake_Handler cb, void∗ userdata) or Fl::awake(void∗ message), but that they never be intermixed. Calling Fl::awake() with no parameters should be safe in either case.

If you have to choose between using the Fl::awake(void∗ message) and Fl::awake(Fl_Awake_Handler cb, void∗ userdata) mechanisms and don't know which to choose, then try the Fl::awake(Fl_Awake_Handler cb, void∗ userdata) method first as it tends to be more powerful in general.

12.4 FLTK multithreaded "lockless programming"

The simple multithreaded examples shown above, using the FLTK lock, work well for many cases where multiple threads are required. However, when that model is extended to more complex programs, it often produces results that the developer did not anticipate.

A typical case might go something like this. A developer creates a program to process a huge data set. The program has a `main()` thread and 7 worker threads and is targeted to run on an 8-core computer. When it runs, the program divides the data between the 7 worker threads, and as they process their share of the data, each thread updates its portion of the GUI with the results, locking and unlocking as they do so.

But when this program runs, it is much slower than expected and the developer finds that only one of the eight CPU cores seems to be utilised, despite there being 8 threads in the program. What happened?

The threads in the program all run as expected, but they end up being serialized (that is, not able to run in parallel) because they all depend on the single FLTK lock. Acquiring (and releasing) that lock has an associated cost, and is a **blocking** action if the lock is already held by any other worker thread or by the `main()` thread.

If the worker threads are acquiring the lock "too often", then the lock will **always** be held **somewhere** and every attempt by any other thread (even `main()`) to lock will cause that other thread (including `main()`) to block. And blocking `main()` also blocks event handling, display refresh...

As a result, only one thread will be running at any given time, and the multithreaded program is effectively reduced to being a (complicated and somewhat less efficient) single thread program.

A "solution" is for the worker threads to lock "less often", such that they do not block each other or the `main()` thread. But judging what constitutes locking "too often" for any given configuration, and hence will block, is a very tricky question. What works well on one machine, with a given graphics card and CPU configuration may behave very differently on another target machine.

There are "interesting" variations on this theme, too: for example it is possible that a "faulty" multithreaded program such as described above will work adequately on a single-core machine (where all threads are inherently serialized anyway and so are less likely to block each other) but then stall or even deadlock in unexpected ways on a multicore machine when the threads do interfere with each other. (I have seen this - it really happens.)

The "better" solution is to avoid using the FLTK lock so far as possible. Instead, the code should be designed so that the worker threads do not update the GUI themselves and therefore never need to acquire the FLTK lock. This would be FLTK multithreaded "lockless programming".

There are a number of ways this can be achieved (or at least approximated) in practice but the most direct approach is for the worker threads to make use of the Fl::awake(Fl_Awake_Handler cb, void∗ userdata) method so that GUI updates can all run in the context of the `main()` thread, alleviating the need for the worker thread to ever lock. The onus is then on the worker threads to manage the `userdata` so that it is delivered safely to the `main()` thread, but there are many ways that can be done.

Note

> Using Fl::awake is not, strictly speaking, entirely "lockless" since the awake handler mechanism incorporates resource locking internally to protect the queue of pending awake messages. These resource locks are held transiently and generally do not trigger the pathological blocking issues described here.

However, aside from using Fl::awake, there are many other ways that a "lockless" design can be implemented, including message passing, various forms of IPC, etc.

If you need high performing multithreaded programming, then take some time to study the options and understand the advantages and disadvantages of each; we can't even begin to scratch the surface of this huge topic here!

And of course occasional, sparse, use of the FLTK lock from worker threads will do no harm; it is "excessive" locking (whatever that might be) that triggers the failing behaviour.

It is always a Good Idea to update the GUI at the lowest rate that is acceptable when processing bulk data (or indeed, in all cases!) Updating at a few frames per second is probably adequate for providing feedback during a long calculation. At the upper limit, anything faster than the frame rate of your monitor and the updates will never even be displayed; why waste CPU computing pixels that you will never show?

12.5 FLTK multithreaded Constraints

FLTK supports multiple platforms, some of which allow only the `main()` thread to handle system events and open or close windows. The safe thing to do is to adhere to the following rules for threads on all operating systems:

- Don't `show()` or `hide()` anything that contains Fl_Window based widgets from a worker thread. This includes any windows, dialogs, file choosers, subwindows or widgets using Fl_Gl_Window. Note that this constraint also applies to non-window widgets that have tooltips, since the tooltip will contain a Fl_Window object. The safe and portable approach is **never** to call `show()` or `hide()` on any widget from the context of a worker thread. Instead you can use the Fl_Awake_Handler variant of Fl::awake() to request the `main()` thread to create, destroy, show or hide the widget on behalf of the worker thread.

- Don't call Fl::run(), Fl::wait(), Fl::flush(), Fl::check() or any related methods that will handle system messages from a worker thread

- Don't intermix use of Fl::awake(Fl_Awake_Handler cb, void∗ userdata) and Fl::awake(void∗ message) calls in the same program as they may interact unpredictably on some platforms; choose one or other style of Fl::awake(<thing>) mechanism and use that. (Intermixing calls to Fl::awake() should be safe with either however.)

- Don't start or cancel timers from a worker thread

- Don't change window decorations or titles from a worker thread

- The `make_current()` method will probably not work well for regular windows, but should always work for a Fl_Gl_Window to allow for high speed rendering on graphics cards with multiple pipelines. Managing thread-safe access to the GL pipelines is left as an exercise for the reader! (And may be target specific...)

See also: Fl::lock(), Fl::unlock(), Fl::awake(), Fl::awake(Fl_Awake_Handler cb, void∗ userdata), Fl::awake(void∗ message), Fl::thread_message().

Chapter 13

Unicode and UTF-8 Support

This chapter explains how FLTK handles international text via Unicode and UTF-8.

Unicode support was only recently added to FLTK and is still incomplete. This chapter is Work in Progress, reflecting the current state of Unicode support.

13.1 About Unicode, ISO 10646 and UTF-8

The summary of Unicode, ISO 10646 and UTF-8 given below is deliberately brief and provides just enough information for the rest of this chapter.

For further information, please see:

- `http://www.unicode.org`
- `http://www.iso.org`
- `http://en.wikipedia.org/wiki/Unicode`
- `http://www.cl.cam.ac.uk/~mgk25/unicode.html`
- `http://www.apps.ietf.org/rfc/rfc3629.html`

The Unicode Standard

The Unicode Standard was originally developed by a consortium of mainly US computer manufacturers and developers of multi-lingual software. It has now become a defacto standard for character encoding and is supported by most of the major computing companies in the world.

Before Unicode, many different systems, on different platforms, had been developed for encoding characters for different languages, but no single encoding could satisfy all languages. Unicode provides access to over 100,000 characters used in all the major languages written today, and is independent of platform and language.

Unicode also provides higher-level concepts needed for text processing and typographic publishing systems, such as algorithms for sorting and comparing text, composite character and text rendering, right-to-left and bi-directional text handling.

Note

There are currently no plans to add this extra functionality to FLTK.

ISO 10646

The International Organisation for Standardization (ISO) had also been trying to develop a single unified character set. Although both ISO and the Unicode Consortium continue to publish their own standards, they have agreed to coordinate their work so that specific versions of the Unicode and ISO 10646 standards are compatible with each other.

The international standard ISO 10646 defines the **Universal Character Set** (UCS) which contains the characters required for almost all known languages. The standard also defines three different implementation levels specifying how these characters can be combined.

Note

> There are currently no plans for handling the different implementation levels or the combining characters in FLTK.

In UCS, characters have a unique numerical code and an official name, and are usually shown using 'U+' and the code in hexadecimal, e.g. U+0041 is the "Latin capital letter A". The UCS characters U+0000 to U+007F correspond to US-ASCII, and U+0000 to U+00FF correspond to ISO 8859-1 (Latin1).

ISO 10646 was originally designed to handle a 31-bit character set from U+00000000 to U+7FFFFFFF, but the current idea is that 21 bits will be sufficient for all future needs, giving characters up to U+10FFFF. The complete character set is sub-divided into *planes*. *Plane 0*, also known as the **Basic Multilingual Plane** (BMP), ranges from U+0000 to U+FFFD and consists of the most commonly used characters from previous encoding standards. Other planes contain characters for specialist applications.

Todo Do we need this info about planes?

The UCS also defines various methods of encoding characters as a sequence of bytes. UCS-2 encodes Unicode characters into two bytes, which is wasteful if you are only dealing with ASCII or Latin1 text, and insufficient if you need characters above U+00FFFF. UCS-4 uses four bytes, which lets it handle higher characters, but this is even more wasteful for ASCII or Latin1.

UTF-8

The Unicode standard defines various UCS Transformation Formats (UTF). UTF-16 and UTF-32 are based on units of two and four bytes. UCS characters requiring more than 16 bits are encoded using "surrogate pairs" in UTF-16.

UTF-8 encodes all Unicode characters into variable length sequences of bytes. Unicode characters in the 7-bit ASCII range map to the same value and are represented as a single byte, making the transformation to Unicode quick and easy.

All UCS characters above U+007F are encoded as a sequence of several bytes. The top bits of the first byte are set to show the length of the byte sequence, and subseqent bytes are always in the range 0x80 to 0xBF. This combination provides some level of synchronisation and error detection.

Unicode range	Byte sequences
`U+00000000 - U+0000007F`	`0xxxxxxx`
`U+00000080 - U+000007FF`	`110xxxxx 10xxxxxx`
`U+00000800 - U+0000FFFF`	`1110xxxx 10xxxxxx 10xxxxxx`
`U+00010000 - U+001FFFFF`	`11110xxx 10xxxxxx 10xxxxxx 10xxxxxx`

`U+00200000 - U+03FFFFFF`	`111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx`
`U+04000000 - U+7FFFFFFF`	`1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx`

Moving from ASCII encoding to Unicode will allow all new FLTK applications to be easily internationalized and used all over the world. By choosing UTF-8 encoding, FLTK remains largely source-code compatible to previous iterations of the library.

13.2 Unicode in FLTK

Todo Work through the code and this documentation to harmonize the [**OksiD**] and [**fltk2**] functions.

FLTK will be entirely converted to Unicode using UTF-8 encoding. If a different encoding is required by the underlying operating system, FLTK will convert the string as needed.

It is important to note that the initial implementation of Unicode and UTF-8 in FLTK involves three important areas:

- provision of Unicode character tables and some simple related functions;
- conversion of char∗ variables and function parameters from single byte per character representation to UTF-8 variable length sequences;
- modifications to the display font interface to accept general Unicode character or UCS code numbers instead of just ASCII or Latin1 characters.

The current implementation of Unicode / UTF-8 in FLTK will impose the following limitations:

- An implementation note in the [**OksiD**] code says that all functions are LIMITED to 24 bit Unicode values, but also says that only 16 bits are really used under linux and win32. **[Can we verify this?]**
- The [**fltk2**] fl_utf8encode() and fl_utf8decode() functions are designed to handle Unicode characters in the range U+000000 to U+10FFFF inclusive, which covers all UTF-16 characters, as specified in RFC 3629. *Note that the user must first convert UTF-16 surrogate pairs to UCS.*
- FLTK will only handle single characters, so composed characters consisting of a base character and floating accent characters will be treated as multiple characters.
- FLTK will only compare or sort strings on a byte by byte basis and not on a general Unicode character basis.
- FLTK will not handle right-to-left or bi-directional text.

Todo Verify 16/24 bit Unicode limit for different character sets? OksiD's code appears limited to 16-bit whereas the FLTK2 code appears to handle a wider set. What about illegal characters? See comments in fl_utf8fromwc() and fl_utf8toUtf16().

13.3 Illegal Unicode and UTF-8 Sequences

Three pre-processor variables are defined in the source code [1] that determine how fl_utf8decode() handles illegal UTF-8 sequences:

- if ERRORS_TO_CP1252 is set to 1 (the default), fl_utf8decode() will assume that a byte sequence starting with a byte in the range 0x80 to 0x9f represents a Microsoft CP1252 character, and will return the value of an equivalent UCS character. Otherwise, it will be processed as an illegal byte value as described below.
- if STRICT_RFC3629 is set to 1 (not the default!) then UTF-8 sequences that correspond to illegal UCS values are treated as errors. Illegal UCS values include those above U+10FFFF, or corresponding to UTF-16 surrogate pairs. Illegal byte values are handled as described below.
- if ERRORS_TO_ISO8859_1 is set to 1 (the default), the illegal byte value is returned unchanged, otherwise 0xFFFD, the Unicode REPLACEMENT CHARACTER, is returned instead.

[1] Since FLTK 1.3.4 you may set these three pre-processor variables on your compile command line with -D"variable=value" (value: 0 or 1) to avoid editing the source code.

fl_utf8encode() is less strict, and only generates the UTF-8 sequence for 0xFFFD, the Unicode REPLACEMENT CHARACTER, if it is asked to encode a UCS value above U+10FFFF.

Many of the [**fltk2**] functions below use fl_utf8decode() and fl_utf8encode() in their own implementation, and are therefore somewhat protected from bad UTF-8 sequences.

The [**OksiD**] fl_utf8len() function assumes that the byte it is passed is the first byte in a UTF-8 sequence, and returns the length of the sequence. Trailing bytes in a UTF-8 sequence will return -1.

- **WARNING:** fl_utf8len() can not distinguish between single bytes representing Microsoft CP1252 characters 0x80-0x9f and those forming part of a valid UTF-8 sequence. You are strongly advised not to use fl_utf8len() in your own code unless you know that the byte sequence contains only valid UTF-8 sequences.
- **WARNING:** Some of the [OksiD] functions below still use fl_utf8len() in their implementations. These may need further validation.

Please see the individual function description for further details about error handling and return values.

13.4 FLTK Unicode and UTF-8 Functions

This section currently provides a brief overview of the functions. For more details, consult the main text for each function via its link.

int fl_utf8locale() **FLTK2**

`fl_utf8locale`() returns true if the "locale" seems to indicate that UTF-8 encoding is used.

It is highly recommended that you change your system so this does return true!

int fl_utf8test(const char ∗src, unsigned len) **FLTK2**

`fl_utf8test`() examines the first `len` bytes of `src`. It returns 0 if there are any illegal UTF-8 sequences; 1 if `src` contains plain ASCII or if `len` is zero; or 2, 3 or 4 to indicate the range of Unicode characters found.

int fl_utf_nb_char(const unsigned char ∗buf, int len) **OksiD**

Returns the number of UTF-8 characters in the first `len` bytes of `buf`.

int fl_unichar_to_utf8_size(Fl_Unichar)

int fl_utf8bytes(unsigned ucs)

Returns the number of bytes needed to encode `ucs` in UTF-8.

int fl_utf8len(char c) **OksiD**

If `c` is a valid first byte of a UTF-8 encoded character sequence, `fl_utf8len()` will return the number of bytes in that sequence. It returns -1 if `c` is not a valid first byte.

unsigned int fl_nonspacing(unsigned int ucs) **OksiD**

Returns true if `ucs` is a non-spacing character.

const char∗ fl_utf8back(const char ∗p, const char ∗start, const char ∗end) **FLTK2**
const char∗ fl_utf8fwd(const char ∗p, const char ∗start, const char ∗end) **FLTK2**

If `p` already points to the start of a UTF-8 character sequence, these functions will return `p`. Otherwise `fl_utf8back()` searches backwards from `p` and `fl_utf8fwd()` searches forwards from `p`, within the `start` and `end` limits, looking for the start of a UTF-8 character.

unsigned int fl_utf8decode(const char ∗p, const char ∗end, int ∗len) **FLTK2**
int fl_utf8encode(unsigned ucs, char ∗buf) **FLTK2**

`fl_utf8decode()` attempts to decode the UTF-8 character that starts at `p` and may not extend past `end`. It returns the Unicode value, and the length of the UTF-8 character sequence is returned via the `len` argument. `fl_utf8encode()` writes the UTF-8 encoding of `ucs` into `buf` and returns the number of bytes in the sequence. See the main documentation for the treatment of illegal Unicode and UTF-8 sequences.

unsigned int fl_utf8froma(char ∗dst, unsigned dstlen, const char ∗src, unsigned srclen) **FLTK2**
unsigned int fl_utf8toa(const char ∗src, unsigned srclen, char ∗dst, unsigned dstlen) **FLTK2**

`fl_utf8froma()` converts a character string containing single bytes per character (i.e. ASCII or ISO-8859-1) into UTF-8. If the `src` string contains only ASCII characters, the return value will be the same as `srclen`.

`fl_utf8toa()` converts a string containing UTF-8 characters into single byte characters. UTF-8 characters that do not correspond to ASCII or ISO-8859-1 characters below 0xFF are replaced with '?'.

Both functions return the number of bytes that would be written, not counting the null terminator. `dstlen` provides a means of limiting the number of bytes written, so setting `dstlen` to zero is a means of measuring how much storage would be needed before doing the real conversion.

char∗ fl_utf2mbcs(const char ∗src) **OksiD**

converts a UTF-8 string to a local multi-byte character string. **[More info required here!]**

unsigned int fl_utf8fromwc(char ∗dst, unsigned dstlen, const wchar_t ∗src, unsigned srclen) **FLTK2**
unsigned int fl_utf8towc(const char ∗src, unsigned srclen, wchar_t ∗dst, unsigned dstlen) **FLTK2**
unsigned int fl_utf8toUtf16(const char ∗src, unsigned srclen, unsigned short ∗dst, unsigned dstlen) **FLTK2**

These routines convert between UTF-8 and `wchar_t` or ”wide character” strings. The difficulty lies in the fact that `sizeof(wchar_t)` is 2 on Windows and 4 on Linux and most other systems. Therefore some ”wide characters” on Windows may be represented as ”surrogate pairs” of more than one `wchar_t`.

`fl_utf8fromwc()` converts from a ”wide character” string to UTF-8. Note that `srclen` is the number of `wchar_t` elements in the source string and on Windows this might be larger than the number of characters. `dstlen` specifies the maximum number of **bytes** to copy, including the null terminator.

`fl_utf8towc()` converts a UTF-8 string into a ”wide character” string. Note that on Windows, some ”wide characters” might result in ”surrogate pairs” and therefore the return value might be more than the number of characters. `dstlen` specifies the maximum number of **wchar_t** elements to copy, including a zero terminating element. **[Is this all worded correctly?]**

`fl_utf8toUtf16()` converts a UTF-8 string into a ”wide character” string using UTF-16 encoding to handle the ”surrogate pairs” on Windows. `dstlen` specifies the maximum number of **wchar_t** elements to copy, including a zero terminating element. **[Is this all worded correctly?]**

These routines all return the number of elements that would be required for a full conversion of the `src` string, including the zero terminator. Therefore setting `dstlen` to zero is a way of measuring how much storage would be needed before doing the real conversion.

unsigned int fl_utf8from_mb(char ∗dst, unsigned dstlen, const char ∗src, unsigned srclen) **FLTK2**
unsigned int fl_utf8to_mb(const char ∗src, unsigned srclen, char ∗dst, unsigned dstlen) **FLTK2**

These functions convert between UTF-8 and the locale-specific multi-byte encodings used on some systems for filenames, etc. If fl_utf8locale() returns true, these functions don’t do anything useful. **[Is this all worded correctly?]**

int fl_tolower(unsigned int ucs) **OksiD**
int fl_toupper(unsigned int ucs) **OksiD**
int fl_utf_tolower(const unsigned char ∗str, int len, char ∗buf) **OksiD**
int fl_utf_toupper(const unsigned char ∗str, int len, char ∗buf) **OksiD**

`fl_tolower()` and `fl_toupper()` convert a single Unicode character from upper to lower case, and vice versa. `fl_utf_tolower()` and `fl_utf_toupper()` convert a string of bytes, some of which may be multi-byte UTF-8 encodings of Unicode characters, from upper to lower case, and vice versa.

Warning: to be safe, `buf` length must be at least `3*len` [for 16-bit Unicode]

int fl_utf_strcasecmp(const char ∗s1, const char ∗s2) **OksiD**
int fl_utf_strncasecmp(const char ∗s1, const char ∗s2, int n) **OksiD**

`fl_utf_strcasecmp()` is a UTF-8 aware string comparison function that converts the strings to lower case Unicode as part of the comparison. `flt_utf_strncasecmp()` only compares the first `n` characters [bytes?]

13.5 FLTK Unicode Versions of System Calls

- int fl_access(const char∗ f, int mode) **OksiD**
- int fl_chmod(const char∗ f, int mode) **OksiD**
- int fl_execvp(const char∗ file, char∗ const∗ argv) **OksiD**
- FILE∗ fl_fopen(cont char∗ f, const char∗ mode) **OksiD**
- char∗ fl_getcwd(char∗ buf, int maxlen) **OksiD**
- char∗ fl_getenv(const char∗ name) **OksiD**
- char fl_make_path(const char∗ path) - returns char ? **OksiD**
- void fl_make_path_for_file(const char∗ path) **OksiD**
- int fl_mkdir(const char∗ f, int mode) **OksiD**
- int fl_open(const char∗ f, int o, ...) **OksiD**
- int fl_rename(const char∗ f, const char∗ t) **OksiD**
- int fl_rmdir(const char∗ f) **OksiD**
- int fl_stat(const char∗ path, struct stat∗ buffer) **OksiD**
- int fl_system(const char∗ f) **OksiD**
- int fl_unlink(const char∗ f) **OksiD**

TODO:

- more doc on unicode, add links
- write something about filename encoding on OS X...
- explain the fl_utf8_... commands
- explain issues with Fl_Preferences
- why FLTK has no Fl_String class

Chapter 14

FLTK Enumerations

Note

> This file is not actively maintained any more, but is left here as a reference, until the doxygen documentation is completed.

See Also

> FL/Enumerations.H.

This appendix lists the enumerations provided in the <FL/Enumerations.H> header file, organized by section. Constants whose value are zero are marked with "(0)", this is often useful to know when programming.

14.1 Version Numbers

The FLTK version number is stored in a number of compile-time constants:

- FL_MAJOR_VERSION - The major release number, currently 1
- FL_MINOR_VERSION - The minor release number, currently 3
- FL_PATCH_VERSION - The patch release number, currently 4
- FL_VERSION - [Deprecated] A combined floating-point version number for the major, minor, and patch release numbers, currently 1.0304
- FL_API_VERSION - A combined integer version number for the major, minor, and patch release numbers, currently 10304 (use this instead of FL_VERSION, if possible)
- FL_ABI_VERSION - A combined integer version number for the application binary interface (ABI) major, minor, and patch release numbers, currently 10300 (default)

Note

> The ABI version (FL_ABI_VERSION) is usually constant throughout one major/minor release version, for instance 10300 if FL_API_VERSION is 10304. Hence the ABI is constant if only the patch version is changed. You can change this with configure or CMake though if you want the latest enhancements (called "ABI features", see CHANGES).

14.2 Events

Events are identified by an Fl_Event enumeration value. The following events are currently defined:

- FL_NO_EVENT - No event (or an event fltk does not understand) occurred (0).
- FL_PUSH - A mouse button was pushed.
- FL_RELEASE - A mouse button was released.
- FL_ENTER - The mouse pointer entered a widget.
- FL_LEAVE - The mouse pointer left a widget.
- FL_DRAG - The mouse pointer was moved with a button pressed.
- FL_FOCUS - A widget should receive keyboard focus.
- FL_UNFOCUS - A widget loses keyboard focus.
- FL_KEYBOARD - A key was pressed.
- FL_CLOSE - A window was closed.
- FL_MOVE - The mouse pointer was moved with no buttons pressed.
- FL_SHORTCUT - The user pressed a shortcut key.
- FL_DEACTIVATE - The widget has been deactivated.
- FL_ACTIVATE - The widget has been activated.
- FL_HIDE - The widget has been hidden.
- FL_SHOW - The widget has been shown.
- FL_PASTE - The widget should paste the contents of the clipboard.
- FL_SELECTIONCLEAR - The widget should clear any selections made for the clipboard.
- FL_MOUSEWHEEL - The horizontal or vertical mousewheel was turned.
- FL_DND_ENTER - The mouse pointer entered a widget dragging data.
- FL_DND_DRAG - The mouse pointer was moved dragging data.
- FL_DND_LEAVE - The mouse pointer left a widget still dragging data.
- FL_DND_RELEASE - Dragged data is about to be dropped.
- FL_SCREEN_CONFIGURATION_CHANGED - The screen configuration (number, positions) was changed.
- FL_FULLSCREEN - The fullscreen state of the window has changed.

14.3 Callback "When" Conditions

The following constants determine when a callback is performed:

- FL_WHEN_NEVER - Never call the callback (0).
- FL_WHEN_CHANGED - Do the callback only when the widget value changes.
- FL_WHEN_NOT_CHANGED - Do the callback whenever the user interacts with the widget.
- FL_WHEN_RELEASE - Do the callback when the button or key is released and the value changes.
- FL_WHEN_ENTER_KEY - Do the callback when the user presses the ENTER key and the value changes.
- FL_WHEN_RELEASE_ALWAYS - Do the callback when the button or key is released, even if the value doesn't change.
- FL_WHEN_ENTER_KEY_ALWAYS - Do the callback when the user presses the ENTER key, even if the value doesn't change.

14.4 Fl::event_button() Values

The following constants define the button numbers for FL_PUSH and FL_RELEASE events:

- FL_LEFT_MOUSE - the left mouse button
- FL_MIDDLE_MOUSE - the middle mouse button
- FL_RIGHT_MOUSE - the right mouse button

14.5 Fl::event_key() Values

The following constants define the non-ASCII keys on the keyboard for FL_KEYBOARD and FL_SHORTCUT events:

- FL_Button - A mouse button; use `Fl_Button + n` for mouse button `n`.
- FL_BackSpace - The backspace key.
- FL_Tab - The tab key.
- FL_Enter - The enter key.
- FL_Pause - The pause key.
- FL_Scroll_Lock - The scroll lock key.
- FL_Escape - The escape key.
- FL_Home - The home key.
- FL_Left - The left arrow key.
- FL_Up - The up arrow key.
- FL_Right - The right arrow key.
- FL_Down - The down arrow key.

- FL_Page_Up - The page-up key.
- FL_Page_Down - The page-down key.
- FL_End - The end key.
- FL_Print - The print (or print-screen) key.
- FL_Insert - The insert key.
- FL_Menu - The menu key.
- FL_Num_Lock - The num lock key.
- FL_KP - One of the keypad numbers; use `FL_KP + n` for number `n`.
- FL_KP_Enter - The enter key on the keypad.
- FL_F - One of the function keys; use `FL_F + n` for function key `n`.
- FL_Shift_L - The lefthand shift key.
- FL_Shift_R - The righthand shift key.
- FL_Control_L - The lefthand control key.
- FL_Control_R - The righthand control key.
- FL_Caps_Lock - The caps lock key.
- FL_Meta_L - The left meta/Windows key.
- FL_Meta_R - The right meta/Windows key.
- FL_Alt_L - The left alt key.
- FL_Alt_R - The right alt key.
- FL_Delete - The delete key.

14.6 Fl::event_state() Values

The following constants define bits in the Fl::event_state() value:

- FL_SHIFT - One of the shift keys is down.
- FL_CAPS_LOCK - The caps lock is on.
- FL_CTRL - One of the ctrl keys is down.
- FL_ALT - One of the alt keys is down.
- FL_NUM_LOCK - The num lock is on.
- FL_META - One of the meta/Windows keys is down.
- FL_COMMAND - An alias for FL_CTRL on WIN32 and X11, or FL_META on MacOS X.
- FL_SCROLL_LOCK - The scroll lock is on.
- FL_BUTTON1 - Mouse button 1 is pushed.
- FL_BUTTON2 - Mouse button 2 is pushed.
- FL_BUTTON3 - Mouse button 3 is pushed.
- FL_BUTTONS - Any mouse button is pushed.
- FL_BUTTON(n) - Mouse button `n` (where `n > 0`) is pushed.

14.7 Alignment Values

The following constants define bits that can be used with Fl_Widget::align() to control the positioning of the label:

- FL_ALIGN_CENTER - The label is centered (0).
- FL_ALIGN_TOP - The label is top-aligned.
- FL_ALIGN_BOTTOM - The label is bottom-aligned.
- FL_ALIGN_LEFT - The label is left-aligned.
- FL_ALIGN_RIGHT - The label is right-aligned.
- FL_ALIGN_CLIP - The label is clipped to the widget.
- FL_ALIGN_WRAP - The label text is wrapped as needed.
- FL_ALIGN_TOP_LEFT - The label appears at the top of the widget, aligned to the left.
- FL_ALIGN_TOP_RIGHT - The label appears at the top of the widget, aligned to the right.
- FL_ALIGN_BOTTOM_LEFT - The label appears at the bottom of the widget, aligned to the left.
- FL_ALIGN_BOTTOM_RIGHT - The label appears at the bottom of the widget, aligned to the right.
- FL_ALIGN_LEFT_TOP - The label appears to the left of the widget, aligned at the top. Outside labels only.
- FL_ALIGN_RIGHT_TOP - The label appears to the right of the widget, aligned at the top. Outside labels only.
- FL_ALIGN_LEFT_BOTTOM - The label appears to the left of the widget, aligned at the bottom. Outside labels only.
- FL_ALIGN_RIGHT_BOTTOM - The label appears to the right of the widget, aligned at the bottom. Outside labels only.
- FL_ALIGN_INSIDE - 'or' this with other values to put label inside the widget.
- FL_ALIGN_TEXT_OVER_IMAGE - Label text will appear above the image.
- FL_ALIGN_IMAGE_OVER_TEXT - Label text will be below the image.
- FL_ALIGN_IMAGE_NEXT_TO_TEXT - The image will appear to the left of the text.
- FL_ALIGN_TEXT_NEXT_TO_IMAGE - The image will appear to the right of the text.
- FL_ALIGN_IMAGE_BACKDROP - The image will be used as a background for the widget.

14.8 Fonts

The following constants define the standard FLTK fonts:

- FL_HELVETICA - Helvetica (or Arial) normal (0).
- FL_HELVETICA_BOLD - Helvetica (or Arial) bold.
- FL_HELVETICA_ITALIC - Helvetica (or Arial) oblique.
- FL_HELVETICA_BOLD_ITALIC - Helvetica (or Arial) bold-oblique.

- FL_COURIER - Courier normal.
- FL_COURIER_BOLD - Courier bold.
- FL_COURIER_ITALIC - Courier italic.
- FL_COURIER_BOLD_ITALIC - Courier bold-italic.
- FL_TIMES - Times roman.
- FL_TIMES_BOLD - Times bold.
- FL_TIMES_ITALIC - Times italic.
- FL_TIMES_BOLD_ITALIC - Times bold-italic.
- FL_SYMBOL - Standard symbol font.
- FL_SCREEN - Default monospaced screen font.
- FL_SCREEN_BOLD - Default monospaced bold screen font.
- FL_ZAPF_DINGBATS - Zapf-dingbats font.

14.9 Colors

The Fl_Color enumeration type holds a FLTK color value. Colors are either 8-bit indexes into a `virtual colormap` or 24-bit RGB color values. Color indices occupy the lower 8 bits of the value, while RGB colors occupy the upper 24 bits, for a byte organization of RGBI.

14.9.1 Color Constants

Constants are defined for the user-defined foreground and background colors, as well as specific colors and the start of the grayscale ramp and color cube in the `virtual colormap`. Inline functions are provided to retrieve specific grayscale, color cube, or RGB color values.

The following color constants can be used to access the user-defined colors:

- FL_BACKGROUND_COLOR - the default background color
- FL_BACKGROUND2_COLOR - the default background color for text, list, and valuator widgets
- FL_FOREGROUND_COLOR - the default foreground color (0) used for labels and text
- FL_INACTIVE_COLOR - the inactive foreground color
- FL_SELECTION_COLOR - the default selection/highlight color

The following color constants can be used to access the colors from the FLTK standard color cube:

- FL_BLACK
- FL_BLUE
- FL_CYAN
- FL_DARK_BLUE
- FL_DARK_CYAN
- FL_DARK_GREEN

- FL_DARK_MAGENTA
- FL_DARK_RED
- FL_DARK_YELLOW
- FL_GREEN
- FL_MAGENTA
- FL_RED
- FL_WHITE
- FL_YELLOW

The following are named values within the standard grayscale:

- FL_GRAY0
- FL_DARK3
- FL_DARK2
- FL_DARK1
- FL_LIGHT1
- FL_LIGHT2
- FL_LIGHT3

The inline methods for getting a grayscale, color cube, or RGB color value are described in the Colors section of the Drawing Things in FLTK chapter.

14.10 Cursors

The following constants define the mouse cursors that are available in FLTK. The double-headed arrows are bitmaps provided by FLTK on X, the others are provided by system-defined cursors.

- FL_CURSOR_DEFAULT - the default cursor, usually an arrow (0)
- FL_CURSOR_ARROW - an arrow pointer
- FL_CURSOR_CROSS - crosshair
- FL_CURSOR_WAIT - watch or hourglass
- FL_CURSOR_INSERT - I-beam
- FL_CURSOR_HAND - hand (uparrow on MSWindows)
- FL_CURSOR_HELP - question mark
- FL_CURSOR_MOVE - 4-pointed arrow
- FL_CURSOR_NS - up/down arrow
- FL_CURSOR_WE - left/right arrow
- FL_CURSOR_NWSE - diagonal arrow
- FL_CURSOR_NESW - diagonal arrow
- FL_CURSOR_NONE - invisible

14.11 FD ”When” Conditions

- FL_READ - Call the callback when there is data to be read.
- FL_WRITE - Call the callback when data can be written without blocking.
- FL_EXCEPT - Call the callback if an exception occurs on the file.

14.12 Damage Masks

The following damage mask bits are used by the standard FLTK widgets:

- FL_DAMAGE_CHILD - A child needs to be redrawn.
- FL_DAMAGE_EXPOSE - The window was exposed.
- FL_DAMAGE_SCROLL - The Fl_Scroll widget was scrolled.
- FL_DAMAGE_OVERLAY - The overlay planes need to be redrawn.
- FL_DAMAGE_USER1 - First user-defined damage bit.
- FL_DAMAGE_USER2 - Second user-defined damage bit.
- FL_DAMAGE_ALL - Everything needs to be redrawn.

Chapter 15

GLUT Compatibility

This appendix describes the GLUT compatibility header file supplied with FLTK.

FLTK's GLUT compatibility is based on the original GLUT 3.7 and the follow-on FreeGLUT 2.4.0 libraries.

15.1 Using the GLUT Compatibility Header File

You should be able to compile existing GLUT source code by including <`FL/glut.H`> instead of <`GL/glut.h`>. This can be done by editing the source, by changing the `-I` switches to the compiler, or by providing a symbolic link from `GL/glut.h` to `FL/glut.H`.

All files calling GLUT procedures must be compiled with C++. You may have to alter them slightly to get them to compile without warnings, and you may have to rename them to get make to use the C++ compiler.

You must link with the FLTK library. Most of `FL/glut.H` is inline functions. You should take a look at it (and maybe at `test/glpuzzle.cxx` in the FLTK source) if you are having trouble porting your GLUT program.

This has been tested with most of the demo programs that come with the GLUT and FreeGLUT distributions.

15.2 Known Problems

The following functions and/or arguments to functions are missing, and you will have to replace them or comment them out for your code to compile:

- `glutGet(GLUT_ELAPSED_TIME)`
- `glutGet(GLUT_SCREEN_HEIGHT_MM)`
- `glutGet(GLUT_SCREEN_WIDTH_MM)`
- `glutGet(GLUT_WINDOW_NUM_CHILDREN)`
- `glutInitDisplayMode(GLUT_LUMINANCE)`
- `glutKeyboardUpFunc`(void(∗callback)(unsigned char key, int x, int y))
- `glutLayerGet(GLUT_HAS_OVERLAY)`
- `glutLayerGet(GLUT_LAYER_IN_USE)`
- `glutPushWindow()`

- `glutSetColor()`, `glutGetColor()`, `glutCopyColormap()`
- `glutVideoResize()` missing.
- `glutWarpPointer()`
- `glutWindowStatusFunc()`
- Spaceball, buttonbox, dials, and tablet functions

Most of the symbols/enumerations have different values than GLUT uses. This will break code that relies on the actual values. The only symbols guaranteed to have the same values are true/false pairs like GLUT-_DOWN and GLUT_UP, mouse buttons GLUT_LEFT_BUTTON, GLUT_MIDDLE_BUTTON, GLUT_R-IGHT_BUTTON, and GLUT_KEY_F1 thru GLUT_KEY_F12.

The strings passed as menu labels are not copied.

`glutPostRedisplay()` does not work if called from inside a display function. You must use `glutIdleFunc()` if you want your display to update continuously.

`glutSwapBuffers()` does not work from inside a display function. This is on purpose, because FLTK swaps the buffers for you.

`glutUseLayer()` does not work well, and should only be used to initialize transformations inside a resize callback. You should redraw overlays by using `glutOverlayDisplayFunc()`.

Overlays are cleared before the overlay display function is called. `glutLayerGet(GLUT_OVERLA-Y_DAMAGED)` always returns true for compatibility with some GLUT overlay programs. You must rewrite your code so that `gl_color()` is used to choose colors in an overlay, or you will get random overlay colors.

`glutSetCursor(GLUT_CURSOR_FULL_CROSSHAIR)` just results in a small crosshair.

The fonts used by `glutBitmapCharacter()` and `glutBitmapWidth()` may be different.

`glutInit(argc,argv)` will consume different switches than GLUT does. It accepts the switches recognized by Fl::args(), and will accept any abbreviation of these switches (such as "-di" for "-display").

15.3 Mixing GLUT and FLTK Code

You can make your GLUT window a child of a Fl_Window with the following scheme. The biggest trick is that GLUT insists on a call to `show()` the window at the point it is created, which means the Fl_Window parent window must already be shown.

- Don't call `glutInit()`.
- Create your Fl_Window, and any FLTK widgets. Leave a blank area in the window for your GLUT window.
- `show()` the Fl_Window. Perhaps call `show(argc,argv)`.
- Call `window->begin()` so that the GLUT window will be automatically added to it.
- Use `glutInitWindowSize()` and `glutInitWindowPosition()` to set the location in the parent window to put the GLUT window.
- Put your GLUT code next. It probably does not need many changes. Call `window->end()` immediately after the `glutCreateWindow()`!
- You can call either `glutMainLoop()`, Fl::run(), or loop calling Fl::wait() to run the program.

15.4 class Fl_Glut_Window

15.4.1 Class Hierarchy

```
Fl_Gl_Window
   |
   +----Fl_Glut_Window
```

15.4.2 Include Files

```
#include <FL/glut.H>
```

15.4.3 Description

Each GLUT window is an instance of this class. You may find it useful to manipulate instances directly rather than use GLUT window id's. These may be created without opening the display, and thus can fit better into FLTK's method of creating windows.

The current GLUT window is available in the global variable `glut_window`.

`new Fl_Glut_Window(...)` is the same as `glutCreateWindow()` except it does not `show()` the window or make the window current.

`window->make_current()` is the same as `glutSetWindow(number)`. If the window has not had `show()` called on it yet, some functions that assume an OpenGL context will not work. If you do `show()` the window, call `make_current()` again to set the context.

`~Fl_Glut_Window()` is the same as `glutDestroyWindow()`.

15.4.4 Members

The Fl_Glut_Window class contains several public members that can be altered directly:

member	description
display	A pointer to the function to call to draw the normal planes.
entry	A pointer to the function to call when the mouse moves into or out of the window.
keyboard	A pointer to the function to call when a regular key is pressed.
menu[3]	The menu to post when one of the mouse buttons is pressed.
mouse	A pointer to the function to call when a button is pressed or released.
motion	A pointer to the function to call when the mouse is moved with a button down.
overlaydisplay	A pointer to the function to call to draw the overlay planes.
passivemotion	A pointer to the function to call when the mouse is moved with no buttons down.
reshape	A pointer to the function to call when the window is resized.
special	A pointer to the function to call when a special key is pressed.
visibility	A pointer to the function to call when the window is iconified or restored (made visible.)

15.4.5 Methods

Fl_Glut_Window::Fl_Glut_Window(int x, int y, int w, int h, const char ∗title = 0)
Fl_Glut_Window::Fl_Glut_Window(int w, int h, const char ∗title = 0)

The first constructor takes 4 int arguments to create the window with a preset position and size. The second constructor with 2 arguments will create the window with a preset size, but the window manager will choose the position according to its own whims.

virtual Fl_Glut_Window::∼Fl_Glut_Window()

Destroys the GLUT window.

void Fl_Glut_Window::make_current()

Switches all drawing functions to the GLUT window.

Chapter 16

Forms Compatibility

This appendix describes the Forms compatibility included with FLTK.

Warning: The Forms compatility is deprecated and no longer maintained in FLTK1, and is likely to be removed completely after the next official release.

16.1 Importing Forms Layout Files

FLUID can read the `.fd` files put out by all versions of Forms and XForms fdesign. However, it will mangle them a bit, but it prints a warning message about anything it does not understand. FLUID cannot write fdesign files, so you should save to a new name so you don't write over the old one.

You will need to edit your main code considerably to get it to link with the output from FLUID. If you are not interested in this you may have more immediate luck with the forms compatibility header, <FL/forms.H>.

16.2 Using the Compatibility Header File

You should be able to compile existing Forms or XForms source code by changing the include directory switch to your compiler so that the `forms.h` file supplied with FLTK is included. The `forms.h` file simply pulls in <FL/forms.H> so you don't need to change your source code. Take a look at <FL/forms.-H> to see how it works, but the basic trick is lots of inline functions. Most of the XForms demo programs work without changes.

You will also have to compile your Forms or XForms program using a C++ compiler. The FLTK library does not provide C bindings or header files.

Although FLTK was designed to be compatible with the GL Forms library (version 0.3 or so), XForms has bloated severely and its interface is X-specific. Therefore, XForms compatibility is no longer a goal of FLTK. Compatibility was limited to things that were free, or that would add code that would not be linked in if the feature is unused, or that was not X-specific.

To use any new features of FLTK, you should rewrite your code to not use the inline functions and instead use "pure" FLTK. This will make it a lot cleaner and make it easier to figure out how to call the FLTK functions. Unfortunately this conversion is harder than expected and even Digital Domain's inhouse code still uses `forms.H` a lot.

16.3 Problems You Will Encounter

Many parts of XForms use X-specific structures like `XEvent` in their interface. I did not emulate these! Unfortunately these features (such as the "canvas" widget) are needed by most large programs. You will need to rewrite these to use FLTK subclasses.

Fl_Free widgets emulate the *old* Forms "free" widget. It may be useful for porting programs that change the `handle()` function on widgets, but you will still need to rewrite things.

Fl_Timer widgets are provided to emulate the XForms timer. These work, but are quite inefficient and inaccurate compared to using Fl::add_timeout().

All instance variables are hidden. If you directly refer to the `x`, `y`, `w`, `h`, `label`, or other fields of your Forms widgets you will have to add empty parenthesis after each reference. The easiest way to do this is to globally replace `"->x"` with `"->x()"`, etc. Replace `"boxtype"` with `"box()"`.

`const char *` arguments to most FLTK methods are simply stored, while Forms would `strdup()` the passed string. This is most noticeable with the label of widgets. Your program must always pass static data such as a string constant or malloc'd buffer to `label()`. If you are using labels to display program output you may want to try the Fl_Output widget.

The default fonts and sizes are matched to the older GL version of Forms, so all labels will draw somewhat larger than an XForms program does.

fdesign outputs a setting of a "fdui" instance variable to the main window. I did not emulate this because I wanted all instance variables to be hidden. You can store the same information in the `user_data()` field of a window. To do this, search through the fdesign output for all occurrences of `"->fdui"` and edit to use `"->user_data()"` instead. This will require casts and is not trivial.

The prototype for the functions passed to `fl_add_timeout()` and `fl_set_idle_callback()` callback are different.

All the following XForms calls are missing:

- `FL_REVISION`, `fl_library_version()`
- `FL_RETURN_DBLCLICK` (use Fl::event_clicks())
- `fl_add_signal_callback()`
- `fl_set_form_atactivate() fl_set_form_atdeactivate()`
- `fl_set_form_property()`
- `fl_set_app_mainform()`, `fl_get_app_mainform()`
- `fl_set_form_minsize()`, `fl_set_form_maxsize()`
- `fl_set_form_event_cmask()`, `fl_get_form_event_cmask()`
- `fl_set_form_dblbuffer()`, `fl_set_object_dblbuffer()` (use an Fl_Double_Window instead)
- `fl_adjust_form_size()`
- `fl_register_raw_callback()`
- `fl_set_object_bw()`, `fl_set_border_width()`
- `fl_set_object_resize()`, `fl_set_object_gravity()`
- `fl_set_object_shortcutkey()`
- `fl_set_object_automatic()`
- `fl_get_object_bbox()` (maybe FLTK should do this)
- `fl_set_object_prehandler()`, `fl_set_object_posthandler()`
- `fl_enumerate_fonts()`
- Most drawing functions
- `fl_set_coordunit()` (FLTK uses pixels all the time)

- `fl_ringbell()`
- `fl_gettime()`
- `fl_win*()` (all these functions)
- `fl_initialize(argc,argv,x,y,z)` ignores last 3 arguments
- `fl_read_bitmapfile()`, `fl_read_pixmapfile()`
- `fl_addto_browser_chars()`
- `FL_MENU_BUTTON` just draws normally
- `fl_set_bitmapbutton_file()`, `fl_set_pixmapbutton_file()`
- `FL_CANVAS` objects
- `FL_DIGITAL_CLOCK` (comes out analog)
- `fl_create_bitmap_cursor()`, `fl_set_cursor_color()`
- `fl_set_dial_angles()`
- `fl_show_oneliner()`
- `fl_set_choice_shortcut(a,b,c)`
- command log
- Only some of file selector is emulated
- `FL_DATE_INPUT`
- `fl_pup*()` (all these functions)
- textbox object (should be easy but I had no sample programs)
- xyplot object

16.4 Additional Notes

These notes were written for porting programs written with the older IRISGL version of Forms. Most of these problems are the same ones encountered when going from old Forms to XForms:

Does Not Run In Background

The IRISGL library always forked when you created the first window, unless "foreground()" was called. FLTK acts like "foreground()" is called all the time. If you really want the fork behavior do "if (fork()) exit(0)" right at the start of your program.

You Cannot Use IRISGL Windows or fl_queue

If a Forms (not XForms) program if you wanted your own window for displaying things you would create a IRISGL window and draw in it, periodically calling Forms to check if the user hit buttons on the panels. If the user did things to the IRISGL window, you would find this out by having the value FL_EVENT returned from the call to Forms.

None of this works with FLTK. Nor will it compile, the necessary calls are not in the interface.

You have to make a subclass of Fl_Gl_Window and write a `draw()` method and `handle()` method. This may require anywhere from a trivial to a major rewrite.

If you draw into the overlay planes you will have to also write a `draw_overlay()` method and call `redraw_overlay()` on the OpenGL window.

One easy way to hack your program so it works is to make the `draw()` and `handle()` methods on your window set some static variables, storing what event happened. Then in the main loop of your program, call Fl::wait() and then check these variables, acting on them as though they are events read from `fl_queue`.

You Must Use OpenGL to Draw Everything

The file <FL/gl.h> defines replacements for a lot of IRISGL calls, translating them to OpenGL. There are much better translators available that you might want to investigate.

You Cannot Make Forms Subclasses

Programs that call `fl_make_object` or directly setting the handle routine will not compile. You have to rewrite them to use a subclass of Fl_Widget. It is important to note that the `handle()` method is not exactly the same as the `handle()` function of Forms. Where a Forms `handle()` returned non-zero, your `handle()` must call `do_callback()`. And your `handle()` must return non-zero if it "understood" the event.

An attempt has been made to emulate the "free" widget. This appears to work quite well. It may be quicker to modify your subclass into a "free" widget, since the "handle" functions match.

If your subclass draws into the overlay you are in trouble and will have to rewrite things a lot.

You Cannot Use <device.h>

If you have written your own "free" widgets you will probably get a lot of errors about "getvaluator". You should substitute:

Forms	FLTK
MOUSE_X	Fl::event_x_root()
MOUSE_Y	Fl::event_y_root()
LEFTSHIFTKEY,RIGHTSHIFTKEY	Fl::event_shift()
CAPSLOCKKEY	Fl::event_capslock()
LEFTCTRLKEY,RIGHTCTRLKEY	Fl::event_ctrl()
LEFTALTKEY,RIGHTALTKEY	Fl::event_alt()
MOUSE1,RIGHTMOUSE	Fl::event_state()
MOUSE2,MIDDLEMOUSE	Fl::event_state()
MOUSE3,LEFTMOUSE	Fl::event_state()

Anything else in `getvaluator` and you are on your own...

Font Numbers Are Different

The ”style” numbers have been changed because I wanted to insert bold-italic versions of the normal fonts. If you use Times, Courier, or Bookman to display any text you will get a different font out of FLTK. If you are really desperate to fix this use the following code:

```
fl_font_name(3,"*courier-medium-r-no*");
fl_font_name(4,"*courier-bold-r-no*");
fl_font_name(5,"*courier-medium-o-no*");
fl_font_name(6,"*times-medium-r-no*");
fl_font_name(7,"*times-bold-r-no*");
fl_font_name(8,"*times-medium-i-no*");
fl_font_name(9,"*bookman-light-r-no*");
fl_font_name(10,"*bookman-demi-r-no*");
fl_font_name(11,"*bookman-light-i-no*");
```

Chapter 17

Operating System Issues

This appendix describes the operating system specific interfaces in FLTK:

- Accessing the OS Interfaces
- The UNIX (X11) Interface
- The Windows (WIN32) Interface
- The Apple OS X Interface

17.1 Accessing the OS Interfaces

All programs that need to access the operating system specific interfaces must include the following header file:

```
#include <FL/x.H>
```

Despite the name, this header file will define the appropriate interface for your environment. The pages that follow describe the functionality that is provided for each operating system.

WARNING:
The interfaces provided by this header file may change radically in new FLTK releases. Use them only when an existing generic FLTK interface is not sufficient.

17.2 The UNIX (X11) Interface

The UNIX interface provides access to the X Window System state information and data structures.

17.2.1 Handling Other X Events

void Fl::add_handler(int (*f)(int))

Installs a function to parse unrecognized events. If FLTK cannot figure out what to do with an event, it calls each of these functions (most recent first) until one of them returns non-zero. If none of them returns non-zero then the event is ignored.

FLTK calls this for any X events it does not recognize, or X events with a window ID that FLTK does not recognize. You can look at the X event in the `fl_xevent` variable.

The argument is the FLTK event type that was not handled, or zero for unrecognized X events. These handlers are also called for global shortcuts and some other events that the widget they were passed to did not handle, for example `FL_SHORTCUT`.

extern XEvent *fl_xevent

This variable contains the most recent X event.

extern ulong fl_event_time

This variable contains the time stamp from the most recent X event that reported it; not all events do. Many X calls like cut and paste need this value.

Window fl_xid(const Fl_Window *)

Returns the XID for a window, or zero if not `shown()`.

Fl_Window *fl_find(ulong xid)

Returns the Fl_Window that corresponds to the given XID, or `NULL` if not found. This function uses a cache so it is slightly faster than iterating through the windows yourself.

int fl_handle(const XEvent &)

This call allows you to supply the X events to FLTK, which may allow FLTK to cooperate with another toolkit or library. The return value is non-zero if FLTK understood the event. If the window does not belong to FLTK and the `add_handler()` functions all return 0, this function will return false.

Besides feeding events your code should call Fl::flush() periodically so that FLTK redraws its windows.

This function will call the callback functions. It will not return until they complete. In particular, if a callback pops up a modal window by calling fl_ask(), for instance, it will not return until the modal function returns.

17.2.2 Drawing using Xlib

The following global variables are set before Fl_Widget::draw() is called, or by Fl_Window::make_current()-:

```
extern Display *fl_display;
extern Window fl_window;
extern GC fl_gc;
extern int fl_screen;
extern XVisualInfo *fl_visual;
extern Colormap fl_colormap;
```

You must use them to produce Xlib calls. Don't attempt to change them. A typical X drawing call is written like this:

```
XDrawSomething(fl_display, fl_window, fl_gc, ...);
```

Other information such as the position or size of the X window can be found by looking at Fl_Window::current(), which returns a pointer to the Fl_Window being drawn.
unsigned long fl_xpixel(Fl_Color i)
unsigned long fl_xpixel(uchar r, uchar g, uchar b)

Returns the X pixel number used to draw the given FLTK color index or RGB color. This is the X pixel that fl_color() would use.

int fl_parse_color(const char∗ p, uchar& r, uchar& g, uchar& b)

Convert a name into the red, green, and blue values of a color by parsing the X11 color names. On other systems, `fl_parse_color()` can only convert names in hexadecimal encoding, for example `#ff8083`.

extern XFontStruct ∗fl_xfont

Points to the font selected by the most recent fl_font(). This is not necessarily the current font of `fl_gc`, which is not set until fl_draw() is called. If FLTK was compiled with Xft support, `fl_xfont` will usually be 0 and `fl_xftfont` will contain a pointer to the `XftFont` structure instead.

extern void ∗fl_xftfont

If FLTK was compiled with Xft support enabled, `fl_xftfont` points to the xft font selected by the most recent fl_font(). Otherwise it will be 0. `fl_xftfont` should be cast to `XftFont∗`.

17.2.3 Changing the Display, Screen, or X Visual

FLTK uses only a single display, screen, X visual, and X colormap. This greatly simplifies its internal structure and makes it much smaller and faster. You can change which it uses by setting global variables *before the first Fl_Window::show() is called.* You may also want to call Fl::visual(), which is a portable interface to get a full color and/or double buffered visual.

int Fl::display(const char ∗)

Set which X display to use. This actually does `putenv("DISPLAY=...")` so that child programs will display on the same screen if called with `exec()`. This must be done before the display is opened. This call is provided under MacOS and WIN32 but it has no effect.

extern Display ∗fl_display

The open X display. This is needed as an argument to most Xlib calls. Don't attempt to change it! This is `NULL` before the display is opened.

void fl_open_display()

Opens the display. Does nothing if it is already open. This will make sure `fl_display` is non-zero. You should call this if you wish to do X calls and there is a chance that your code will be called before the first `show()` of a window.

This may call Fl::abort() if there is an error opening the display.

void fl_close_display()

This closes the X connection. You do *not* need to call this to exit, and in fact it is faster to not do so! It may be useful to call this if you want your program to continue without the X connection. You cannot open the display again, and probably cannot call any FLTK functions.

extern int fl_screen

Which screen number to use. This is set by `fl_open_display()` to the default screen. You can change it by setting this to a different value immediately afterwards. It can also be set by changing the last number in the Fl::display() string to "host:0.#".

extern XVisualInfo ∗fl_visual
extern Colormap fl_colormap

The visual and colormap that FLTK will use for all windows. These are set by `fl_open_display()` to the default visual and colormap. You can change them before calling `show()` on the first window. Typical code for changing the default visual is:

```
Fl::args(argc, argv); // do this first so $DISPLAY is set
fl_open_display();
fl_visual = find_a_good_visual(fl_display, fl_screen);
if (!fl_visual) Fl::abort("No good visual");
fl_colormap = make_a_colormap(fl_display, fl_visual->visual, fl_visual->depth);
// it is now ok to show() windows:
window->show(argc, argv);
```

17.2.4 Using a Subclass of Fl_Window for Special X Stuff

FLTK can manage an X window on a different screen, visual and/or colormap, you just can't use FLTK's drawing routines to draw into it. But you can write your own `draw()` method that uses Xlib (and/or OpenGL) calls only.

FLTK can also manage XID's provided by other libraries or programs, and call those libraries when the window needs to be redrawn.

To do this, you need to make a subclass of Fl_Window and override some of these virtual functions:
virtual void Fl_Window::show()

If the window is already `shown()` this must cause it to be raised, this can usually be done by calling Fl_Window::show(). If not `shown()` your implementation must call either Fl_X::set_xid() or Fl_X::make_xid().

An example:

```
void MyWindow::show() {
  if (shown()) {Fl_Window::show(); return;}  // you must do this!
  fl_open_display();    // necessary if this is first window
  // we only calculate the necessary visual colormap once:
  static XVisualInfo *visual;
  static Colormap colormap;
  if (!visual) {
    visual = figure_out_visual();
    colormap = XCreateColormap(fl_display, RootWindow(fl_display,fl_screen),
                               vis->visual, AllocNone);
  }
  Fl_X::make_xid(this, visual, colormap);
}
```

`Fl_X *Fl_X::set_xid(Fl_Window*, Window xid)`

Allocate a hidden class called an Fl_X, put the XID into it, and set a pointer to it from the Fl_Window. This causes Fl_Window::shown() to return true.

void Fl_X::make_xid(Fl_Window∗, XVisualInfo∗ = fl_visual, Colormap = fl_colormap)

This static method does the most onerous parts of creating an X window, including setting the label, resize limitations, etc. It then does Fl_X::set_xid() with this new window and maps the window.

virtual void Fl_Window::flush()

This virtual function is called by Fl::flush() to update the window. For FLTK's own windows it does this by setting the global variables `fl_window` and `fl_gc` and then calling the `draw()` method. For your own windows you might just want to put all the drawing code in here.

The X region that is a combination of all `damage()` calls done so far is in `Fl_X::i(this)->region`. If `NULL` then you should redraw the entire window. The undocumented function `fl_clip_region(-XRegion)` will initialize the FLTK clip stack with a region or `NULL` for no clipping. You must set region to `NULL` afterwards as `fl_clip_region()` will own and delete it when done.

If `damage() & FL_DAMAGE_EXPOSE` then only X expose events have happened. This may be useful if you have an undamaged image (such as a backing buffer) around.

Here is a sample where an undamaged image is kept somewhere:

```
void MyWindow::flush() {
  fl_clip_region(Fl_X::i(this)->region);
  Fl_X::i(this)->region = 0;
  if (damage() != 2) {... draw things into backing store ...}
  ... copy backing store to window ...
}
```

virtual void Fl_Window::hide()

Destroy the window server copy of the window. Usually you will destroy contexts, pixmaps, or other resources used by the window, and then call Fl_Window::hide() to get rid of the main window identified by `xid()`. If you override this, you must also override the destructor as shown:

```
void MyWindow::hide() {
  if (mypixmap) {
    XFreePixmap(fl_display,mypixmap);
    mypixmap = 0;
  }
  Fl_Window::hide(); // you must call this
}
```

virtual void Fl_Window::∼Fl_Window()

Because of the way C++ works, if you override `hide()` you *must* override the destructor as well (otherwise only the base class `hide()` is called):

```
MyWindow::~MyWindow() {
  hide();
}
```

Note

Access to the Fl_X hidden class requires to #define FL_INTERNALS before compilation.

17.2.5 Setting the Icon of a Window

FLTK currently supports setting a window's icon **before** it is shown using the Fl_Window::icon() method.
void Fl_Window::icon(const void ∗)

Sets the icon for the window to the passed pointer. You will need to cast the icon `Pixmap` to a `char∗` when calling this method. To set a monochrome icon using a bitmap compiled with your application use:

```
#include "icon.xbm"

fl_open_display(); // needed if display has not been previously opened

Pixmap p = XCreateBitmapFromData(fl_display, DefaultRootWindow(fl_display),
                                 icon_bits, icon_width, icon_height);

window->icon((const void*)p);
```

To use a multi-colored icon, the XPM format and library should be used as follows:

```
#include <X11/xpm.h>
#include "icon.xpm"

fl_open_display();               // needed if display has not been previously opened

Pixmap p, mask;

XpmCreatePixmapFromData(fl_display, DefaultRootWindow(fl_display),
                                 icon_xpm, &p, &mask, NULL);

window->icon((const void *)p);
```

When using the Xpm library, be sure to include it in the list of libraries that are used to link the application (usually "-lXpm").

NOTE:
You must call Fl_Window::show(int argc, char** argv) for the icon to be used. The Fl_Window::show() method does not bind the icon to the window.

17.2.6 X Resources

When the Fl_Window::show(int argc, char** argv) method is called, FLTK looks for the following X resources:

- `background` - The default background color for widgets (color).
- `dndTextOps` - The default setting for drag and drop text operations (boolean).
- `foreground` - The default foreground (label) color for widgets (color).
- `scheme` - The default scheme to use (string).
- `selectBackground` - The default selection color for menus, etc. (color).
- `Text.background` - The default background color for text fields (color).
- `tooltips` - The default setting for tooltips (boolean).
- `visibleFocus` - The default setting for visible keyboard focus on non-text widgets (boolean).

Resources associated with the first window's Fl_Window::xclass() string are queried first, or if no class has been specified then the class "fltk" is used (e.g. `fltk.background`). If no match is found, a global search is done (e.g. `*background`).

17.3 The Windows (WIN32) Interface

The Windows interface provides access to the WIN32 GDI state information and data structures.

17.3.1 Using filenames with non-ASCII characters

In FLTK, all strings, including filenames, are UTF-8 encoded. The utility functions fl_fopen() and fl_open() allow to open files potentially having non-ASCII names in a cross-platform fashion, whereas the standard fopen()/open() functions fail to do so.

17.3.2 Responding to WM_QUIT

FLTK will intercept WM_QUIT messages that are directed towards the thread that runs the main loop. These are converted to SIGTERM signals via `raise()`. This allows you to deal with outside termination requests with the same code on both Windows and UNIX systems. Other processes can send this message via `PostThreadMessage()` in order to request, rather than force your application to terminate.

17.3.3 Handling Other WIN32 Messages

By default a single WNDCLASSEX called "FLTK" is created. All Fl_Window 's are of this class unless you use Fl_Window::xclass(). The window class is created the first time Fl_Window::show() is called.

You can probably combine FLTK with other libraries that make their own WIN32 window classes. The easiest way is to call Fl::wait(), as it will call `DispatchMessage()` for all messages to the other windows. If necessary you can let the other library take over as long as it calls `DispatchMessage()`, but you will have to arrange for the function Fl::flush() to be called regularly so that widgets are updated, timeouts are handled, and the idle functions are called.

extern MSG fl_msg

This variable contains the most recent message read by `GetMessage()`, which is called by Fl::wait(). This may not be the most recent message sent to an FLTK window, because silly WIN32 calls the handle procedures directly for some events (sigh).

void Fl::add_handler(int (*f)(int))

Installs a function to parse unrecognized messages sent to FLTK windows. If FLTK cannot figure out what to do with a message, it calls each of these functions (most recent first) until one of them returns non-zero. The argument passed to the functions is the FLTK event that was not handled or zero for unknown messages. If all the handlers return zero then FLTK calls `DefWindowProc()`.

HWND fl_xid(const Fl_Window *)

Returns the window handle for a Fl_Window, or zero if not `shown()`.

Fl_Window *fl_find(HWND xid)

Returns the Fl_Window that corresponds to the given window handle, or `NULL` if not found. This function uses a cache so it is slightly faster than iterating through the windows yourself.

17.3.4 Drawing Things Using the WIN32 GDI

When the virtual function Fl_Widget::draw() is called, FLTK stores all the extra arguments you need to make a proper GDI call in some global variables:

```
extern HINSTANCE fl_display;
extern HWND fl_window;
extern HDC fl_gc;
COLORREF fl_RGB();
HPEN fl_pen();
HBRUSH fl_brush();
```

These global variables are set before Fl_Widget::draw() is called, or by Fl_Window::make_current(). You can refer to them when needed to produce GDI calls, but don't attempt to change them. The functions return GDI objects for the current color set by fl_color() and are created as needed and cached. A typical GDI drawing call is written like this:

```
DrawSomething(fl_gc, ..., fl_brush());
```

It may also be useful to refer to Fl_Window::current() to get the window's size or position.

17.3.5 Setting the Icon of a Window

FLTK currently supports setting a window's icon *before* it is shown using the Fl_Window::icon() method.
void Fl_Window::icon(const void ∗)

Sets the icon for the window to the passed pointer. You will need to cast the `HICON` handle to a `char*` when calling this method. To set the icon using an icon resource compiled with your application use:

```
window->icon((const void *)LoadIcon(fl_display, MAKEINTRESOURCE(IDI_ICON)));
```

You can also use the `LoadImage()` and related functions to load specific resolutions or create the icon from bitmap data.

NOTE:
You must call Fl_Window::show(int argc, char∗∗ argv) for the icon to be used. The Fl_Window::show() method does not bind the icon to the window.

17.3.6 How to Not Get a MSDOS Console Window

WIN32 has a really stupid mode switch stored in the executables that controls whether or not to make a console window.

To always get a console window you simply create a console application (the "/SUBSYSTEM:CONSOLE" option for the linker). For a GUI-only application create a WIN32 application (the "/SUBSYSTEM:WINDOWS" option for the linker).

FLTK includes a `WinMain()` function that calls the ANSI standard `main()` entry point for you. *This function creates a console window when you use the debug version of the library.*

WIN32 applications without a console cannot write to `stdout` or `stderr`, even if they are run from a console window. Any output is silently thrown away. Additionally, WIN32 applications are run in the background by the console, although you can use "start /wait program" to run them in the foreground.

17.3.7 Known WIN32 Bugs and Problems

The following is a list of known bugs and problems in the WIN32 version of FLTK:

- If a program is deactivated, `Fl::wait()` does not return until it is activated again, even though many events are delivered to the program. This can cause idle background processes to stop unexpectedly. This also happens while the user is dragging or resizing windows or otherwise holding the mouse down. We were forced to remove most of the efficiency FLTK uses for redrawing in order to get windows to update while being moved. This is a design error in WIN32 and probably impossible to get around.
- `Fl_Gl_Window::can_do_overlay()` returns true until the first time it attempts to draw an overlay, and then correctly returns whether or not there is overlay hardware.
- `SetCapture` (used by `Fl::grab()`) doesn't work, and the main window title bar turns gray while menus are popped up.

- Compilation with `gcc 3.4.4` and `-Os` exposes an optimisation bug in gcc. The symptom is that when drawing filled circles only the perimeter is drawn. This can for instance be seen in the symbols demo. Other optimisation options such as -O2 and -O3 seem to work OK. More details can be found in STR#1656

17.4 The Apple OS X Interface

FLTK supports Apple OS X using the Apple Cocoa library. Older versions of MacOS are no longer supported.

Control, Option, and Command Modifier Keys

FLTK maps the Mac 'control' key to `FL_CTRL`, the 'option' key to `FL_ALT` and the 'Apple' key to `FL_META`. Furthermore, `FL_COMMAND` designates the 'Apple' key on Mac OS X and the 'control' key on other platforms. Keyboard events return the key name in Fl::event_key() and the keystroke translation in Fl::event_text(). For example, typing Option-Y on a Mac US keyboard will set `FL_ALT` in Fl::event_state(), set Fl::event_key() to 'y' and return the Yen symbol in Fl::event_text().

Right Click simulation with Ctrl Click

The Apple HIG guidelines indicate applications should support 'Ctrl Click' to simulate 'Right Click' for e.g. context menus, so users with one-button mice and one-click trackpads can still access right-click features. However, paraphrasing `Manolo's comment on the fltk.coredev newsgroup`- :

- *FLTK does /not/ support Ctrl-Click == Right Click itself because Mac OS X event processing doesn't support this at the system level: the system reports left-clicks with the ctrl modifier when the user ctrl-clicks, and OS X system preferences don't allow changing this behavior. Therefore, applications must handle simulation of Right Click with Ctrl Click in the application code.*

Ian MacArthur provided the following handle() method code snippet showing an example of how to do this:

```
    case FL_PUSH:
    {
        int btn = Fl::event_button();
#ifdef __APPLE__
        int ev_state = Fl::event_state();
#endif
        //
        // Context menu can be called up in one of two ways: -
        //     1 - right click, as normally used on Windows and Linux
        //     2 - Ctrl + left click, as sometimes used on Mac
        //
#ifdef __APPLE__
        // On apple, check right click, and ctrl+left click
        if ((btn == FL_RIGHT_MOUSE) || (ev_state == (FL_CTRL |
      FL_BUTTON1)))
#else
        // On other platforms, only check right click as ctrl+left is used for selections
        if (btn == FL_RIGHT_MOUSE)
#endif
        {
            // Did we right click on the object?..
```

There is a thread about this subject on fltk.coredev (Aug 1-14, 2014) entitled "[RFC] Right click emulation for one button mouse on Mac".

Apple "Quit" Event

When the user presses Cmd-Q or requests a termination of the application, OS X will send a "Quit" Apple Event. FLTK handles this event by sending an `FL_CLOSE` event to all open windows. If all windows close, the application will terminate.

Apple "Open" Event

Whenever the user drops a file onto an application icon, OS X generates an Apple Event of the type "Open". You can have FLTK notify you of an Open event by calling the fl_open_callback function.

void fl_open_display()

Opens the display. Does nothing if it is already open. You should call this if you wish to do Cocoa or Quartz calls and there is a chance that your code will be called before the first `show()` of a window.

Window fl_xid(const Fl_Window ∗)

Returns the window reference for an Fl_Window, or `NULL` if the window has not been shown. This reference is a pointer to an instance of the subclass FLWindow of Cocoa's NSWindow class.

Fl_Window ∗fl_find(Window xid)

Returns the Fl_Window that corresponds to the given window reference, or `NULL` if not found.

void fl_mac_set_about(Fl_Callback ∗cb, void ∗user_data, int shortcut)

Attaches the callback `cb` to the "About myprog" item of the system application menu. `cb` will be called with NULL first argument and `user_data` second argument.

Fl_Sys_Menu_Bar class

The Fl_Sys_Menu_Bar class allows to build menu bars that, on Mac OS X, are placed in the system menu bar (at top-left of display), and, on other platforms, at a user-chosen location of a user-chosen window.

17.4.1 Setting the icon of an application

- First, create a .icns file containing several copies of your icon of decreasing sizes. This can be done using the Preview application or the Icon Composer application available in "Graphics Tools for Xcode". To create a high resolution icon file, it is necessary to use the iconutil command-line utility.
- Put your .icns file in the Resources subdirectory of your application bundle.
- Add these two lines to the Info.plist file of your application bundle

```
<key>CFBundleIconFile</key>
<string>foo.icns</string>
```

replacing `foo` by your application name. If you use Xcode, just add your .icns file to your application target.

17.4.2 Drawing Things Using Quartz

All code inside Fl_Widget::draw() is expected to call Quartz drawing functions. The Quartz coordinate system is flipped to match FLTK's coordinate system. The origin for all drawing is in the top left corner of the enclosing Fl_Window. The global variable `fl_gc` (of type `CGContextRef`) is the appropriate Quartz 2D drawing environment. Include FL/x.H to declare the `fl_gc` variable.

17.4.3 Internationalization

All FLTK programs contain an application menu with, e.g., the About xxx, Hide xxx, and Quit xxx items. This menu can be internationalized/localized by any of two means.

- using the Fl_Mac_App_Menu class.
- using the standard Mac OS X localization procedure. Create a language-specific .lproj directory (e.g., `German.lproj`) in the Resources subdirectory of the application bundle. Create therein a `Localizable.strings` file that translates all menu items to this language. The German `Localizable.strings` file, for example, contains:

```
"About %@" = "Über %@";
"Print Front Window"="Frontfenster drucken";
"Services" = "Dienste";
"Hide %@"="%@ ausblenden";
"Hide Others"="Andere ausblenden";
"Show All"="Alle einblenden";
"Quit %@"="%@ beenden";
```

  Set `"Print Front Window" = "";` therein so the application menu doesn't show a "Print Front Window" item. To localize the application name itself, create a file `InfoPlist.strings` in each .lproj directory and put `CFBundleName = "localized name";` in each such file.

17.4.4 OpenGL and 'retina' displays

It is possible to have OpenGL produce graphics at the high pixel resolution allowed by the so-called 'retina' displays present on recent Apple hardware. For this, call

```
Fl::use_high_res_GL(1);
```

before any Fl_Gl_Window is shown. Also, adapt your Fl_Gl_Window::draw() and Fl_Gl_Window::draw_-overlay() methods replacing

```
glViewport(0, 0, w(), h());
```

by

```
glViewport(0, 0, pixel_w(), pixel_h());
```

making use of the Fl_Gl_Window::pixel_w() and Fl_Gl_Window::pixel_h() methods that return the width and height of the GL scene in pixels: if the Fl_Gl_Window is mapped on a retina display, these methods return twice as much as reported by Fl_Widget::w() and Fl_Widget::h(); if it's mapped on a regular display, they return the same values as w() and h(). These methods dynamically change their values if the window is moved into/out from a retina display. If Fl::use_high_res_GL(1) is not called, all Fl_Gl_Window 's are drawn at low resolution. These methods are synonyms of w() and h() on non-Mac OS X platforms, so the source code remains cross-platform.

The Fl_Gl_Window::pixels_per_unit() method is useful when the OpenGL code depends on the pixel dimension of the GL scene. This occurs, e.g., if a window's handle() method uses Fl::event_x() and Fl-::event_y() whose returned values should be multiplied by Fl_Gl_Window::pixels_per_unit() to obtain the adequate pixel units. This method may also be useful, for example, to adjust the width of a line in a high resolution GL scene.

17.4.5 Fl_Double_Window

OS X double-buffers all windows automatically. On OS X, Fl_Window and Fl_Double_Window are handled internally in the same way.

17.4.6 Mac File System Specifics

Resource Forks

FLTK does not access the resource fork of an application. However, a minimal resource fork must be created for OS X applications. Starting with OS X 10.6, resource forks are no longer needed.

Caution (OS X 10.2 and older):
When using UNIX commands to copy or move executables, OS X will NOT copy any resource forks! For copying and moving use CpMac and MvMac respectively. For creating a tar archive, all executables need to be stripped from their Resource Fork before packing, e.g. ”DeRez fluid > fluid.r”. After unpacking the Resource Fork needs to be reattached, e.g. ”Rez fluid.r -o fluid”.

It is advisable to use the Finder for moving and copying and Mac archiving tools like Sit for distribution as they will handle the Resource Fork correctly.

Mac File Paths

FLTK uses UTF-8-encoded UNIX-style filenames and paths.

See Also

Mac OS X-specific symbols

Chapter 18

Migrating Code from FLTK 1.0 to 1.1

This appendix describes the differences between the FLTK 1.0.x and FLTK 1.1.x functions and classes.

18.1 Color Values

Color values are now stored in a 32-bit unsigned integer instead of the unsigned character in 1.0.x. This allows for the specification of 24-bit RGB values or 8-bit FLTK color indices.

FL_BLACK and FL_WHITE now remain black and white, even if the base color of the gray ramp is changed using Fl::background(). FL_DARK3 and FL_LIGHT3 can be used instead to draw a very dark or a very bright background hue.

Widgets use the new color symbols FL_FOREGROUND_COLOR, FL_BACKGROUND_COLOR, FL_BACKGROUND2_COLOR, FL_INACTIVE_COLOR, and FL_SELECTION_COLOR. More details can be found in the chapter FLTK Enumerations.

18.2 Cut and Paste Support

The FLTK clipboard is now broken into two parts - a local selection value and a cut-and-paste value. This allows FLTK to support things like highlighting and replacing text that was previously cut or copied, which makes FLTK applications behave like traditional GUI applications.

18.3 File Chooser

The file chooser in FLTK 1.1.x is significantly different than the one supplied with FLTK 1.0.x. Any code that directly references the old FCB class or members will need to be ported to the new Fl_File_Chooser class.

18.4 Function Names

Some function names have changed from FLTK 1.0.x to 1.1.x in order to avoid name space collisions. You can still use the old function names by defining the FLTK_1_0_COMPAT symbol on the command-line when you compile (-DFLTK_1_0_COMPAT) or in your source, e.g.:

```
#define FLTK_1_0_COMPAT
#include <FL/Fl.H>
#include <FL/Enumerations.H>
#include <FL/filename.H>
```

The following table shows the old and new function names:

Old 1.0.x Name	New 1.1.x Name
contrast()	fl_contrast()
down()	fl_down()
filename_absolute()	fl_filename_absolute()
filename_expand()	fl_filename_expand()
filename_ext()	fl_filename_ext()
filename_isdir()	fl_filename_isdir()
filename_list()	fl_filename_list()
filename_match()	fl_filename_match()
filename_name()	fl_filename_name()
filename_relative()	fl_filename_relative()
filename_setext()	fl_filename_setext()
frame()	fl_frame()
inactive()	fl_inactive()
numericsort()	fl_numericsort()

18.5 Image Support

Image support in FLTK has been significantly revamped in 1.1.x. The Fl_Image class is now a proper base class, with the core image drawing functionality in the Fl_Bitmap, Fl_Pixmap, and Fl_RGB_Image classes.

BMP, GIF, JPEG, PNG, XBM, and XPM image files can now be loaded using the appropriate image classes, and the Fl_Shared_Image class can be used to cache images in memory.

Image labels are no longer provided as an add-on label type. If you use the old `label()` methods on an image, the widget's `image()` method is called to set the image as the label.

Image labels in menu items must still use the old labeltype mechanism to preserve source compatibility.

18.6 Keyboard Navigation

FLTK 1.1.x now supports keyboard navigation and control with all widgets. To restore the old FLTK 1.0.x behavior so that only text widgets get keyboard focus, call the Fl::visible_focus() method to disable it:

```
Fl::visible_focus(0);
```

Chapter 19

Migrating Code from FLTK 1.1 to 1.3

This appendix describes the differences between the FLTK 1.1.x and FLTK 1.3.x functions and classes.

19.1 Migrating From FLTK 1.0

If you want to migrate your code from FLTK 1.0 to FLTK 1.3, then you should first consult Appendix Migrating Code from FLTK 1.0 to 1.1.

19.2 Fl_Scroll Widget

Fl_Scroll::scroll_to(int x, int y) replaces Fl_Scroll::position(int x, int y).

This change was needed because Fl_Scroll::position(int,int) redefined Fl_Widget::position(int,int), but with a completely different function (moving the scrollbars instead of moving the widget).

Please be aware that you need to change your application's code for all Fl_Scroll-derived widgets, if you used Fl_Scroll::position(int x, int y) to position **the scrollbars** (not the widget itself).

The compiler will not detect any errors, because your calls to **position(int x, int y)** will be calling Fl_Widget::position(int x, int y).

19.3 Unicode (UTF-8)

FLTK 1.3 uses Unicode (UTF-8) encoding internally. If you are only using characters in the ASCII range (32-127), there is a high probability that you don't need to modify your code. However, if you use international characters (128-255), encoded as e.g. Windows codepage 1252, ISO-8859-1, ISO-8859-15 or any other encoding, then you will need to update your character string constants and widget input data accordingly.

Please refer to the Unicode and UTF-8 Support chapter for more details.

Note

> It is important that, although your software uses only ASCII characters for input to FLTK widgets, the user may enter non-ASCII characters, and FLTK will return these characters with UTF-8 encoding to your application, e.g. via Fl_Input::value(). You **will** need to re-encode them to **your** (non-UTF-8) encoding, otherwise you might see or print garbage in your data.

19.4 Widget Coordinate Representation

FLTK 1.3 changed all Widget coordinate variables and methods, e.g. Fl_Widget::x(), Fl_Widget::y(), Fl_Widget::w(), Fl_Widget::h(), from short (16-bit) to int (32-bit) representation. This should not affect any existing code, but makes it possible to use bigger scroll areas (e.g. Fl_Scroll widget).

Chapter 20

Developer Information

This chapter describes FLTK development and documentation.

Example

```
/** \file
   Fl_Clock, Fl_Clock_Output widgets. */


/**
  \class Fl_Clock_Output
  \brief This widget can be used to display a program-supplied time.

  The time shown on the clock is not updated. To display the current time,
  use Fl_Clock instead.

  \image html clock.png
  \image latex clock.png "" width=10cm
  \image html round_clock.png
  \image latex clock.png "" width=10cm
  \image html round_clock.png "" width=10cm */

  /**
    Returns the displayed time.
    Returns the time in seconds since the UNIX epoch (January 1, 1970).
    \see value(ulong)
   */
  ulong value() const {return value_;}

/**
  Set the displayed time.
  Set the time in seconds since the UNIX epoch (January 1, 1970).
  \param[in] v seconds since epoch
  \see value()
 */
void Fl_Clock_Output::value(ulong v) {
 [...]
}

/**
  Create an Fl_Clock widget using the given position, size, and label string.
  The default boxtype is \c FL_NO_BOX.
  \param[in] X, Y, W, H position and size of the widget
  \param[in] L widget label, default is no label
 */
Fl_Clock::Fl_Clock(int X, int Y, int W, int H, const char *L)
  : Fl_Clock_Output(X, Y, W, H, L) {}

/**
```

```
  Create an Fl_Clock widget using the given boxtype, position, size, and
  label string.
  \param[in] t boxtype
  \param[in] X, Y, W, H position and size of the widget
  \param[in] L widget label, default is no label
 */
Fl_Clock::Fl_Clock(uchar t, int X, int Y, int W, int H, const char *L)
  : Fl_Clock_Output(X, Y, W, H, L) {
  type(t);
  box(t==FL_ROUND_CLOCK ? FL_NO_BOX : FL_UP_BOX);
}
```

Note

From Duncan: (will be removed later, just for now as a reminder)

I've just added comments for the fl_color_chooser() functions, and in order to keep them and the general Function Reference information for them together, I created a new doxygen group, and used \ingroup in the three comment blocks. This creates a new Modules page (which may not be what we want) with links to it from the File Members and Fl_Color_Chooser.H pages. It needs a bit more experimentation on my part unless someone already knows how this should be handled. (Maybe we can add it to a functions.dox file that defines a functions group and do that for all of the function documentation?)

Update: the trick is not to create duplicate entries in a new group, but to move the function information into the doxygen comments for the class, and use the navigation links provided. Simply using \relatesalso as the first doxygen command in the function's comment puts it in the appropriate place. There is no need to have \defgroup and \ingroup as well, and indeed they don't work. So, to summarize:

```
Gizmo.H
  /** \class Gizmo
      A gizmo that does everything
    */
  class Gizmo {
    etc
  };
  extern int popup_gizmo(...);

Gizmo.cxx:
  /** \relatesalso Gizmo
      Pops up a gizmo dialog with a Gizmo in it
    */
  int popup_gizmo(...);
```

Comments Within Doxygen Comment Blocks

You can use HTML comment statements to embed comments in doxygen comment blocks. These comments will not be visible in the generated document.

```
The following text is a developer comment.
<!-- *** This *** is *** invisible *** -->
This will be visible again.
```

will be shown as:

```
The following text is a developer comment.

This will be visible again.
```

Different Headlines

You can use HTML tags <H1> ... <H4> for headlines with different sizes. As of doxygen 1.8.x there must not be more than three spaces at the beginning of the line for this to work. Currently (doxygen 1.8.6) there seems to be no difference in the font sizes of <H3> and <H4> in the pdf output, whereas the html output uses different font sizes.

```
<H1>Headline in big text (H1)</H1>
<H2>Headline in big text (H2)</H2>
<H3>Headline in big text (H3)</H3>
<H4>Headline in big text (H4)</H4>
```

Headline in big text (H1)

Headline in big text (H2)

Headline in big text (H3)

Headline in big text (H4)

20.1 Non-ASCII Characters

```
Doxygen understands many HTML quoting characters like
&quot;, &uuml;, &ccedil;, &Ccedil;, but not all HTML quoting characters.
```

This will appear in the document:

```
Doxygen understands many HTML quoting characters like
&quot;, &uuml;, &ccedil;, &Ccedil;, but not all HTML quoting characters.
```

For further informations about HTML quoting characters see
http://www.doxygen.org/htmlcmds.html
Alternatively you can use **UTF-8** encoding within Doxygen comments.

20.2 Document Structure

- **page** creates a named page
- **section** creates a named section within that page
- **subsection** creates a named subsection within the current section
- **subsubsection** creates a named subsubsection within the current subsection

All these statements take a ”name” as their first argument, and a title as their second argument. The title can contain spaces.

The page, section, and subsection titles are formatted in blue color and a size like ”<**H1**>”, ”<**H2**>”, and ”<**H3**>”, and ”<**H4**>”, respectively.

By **FLTK documentation convention**, a file like this one with a doxygen documentation chapter has the name ”<**chapter**>**.dox”.** The **page** statement at the top of the page is ”**page** <**chapter**> **This is the title”**. Sections within a documentation page must be called ”<**chapter**>_<**section**>”, where ”<**chapter**>” is the name part of the file, and ”<**section**>” is a unique section name within the page that can be referenced in links. The same for subsections and subsubsections.

These doxygen page and section commands work only in special documentation chapters, not within normal source or header documentation blocks. However, links **from** normal (e.g. class) documentation **to** documentation sections **do work**.

This page has

```
\page development I - Developer Information
```

at its top.

This section is

```
\section development_structure Document Structure
```

The following section is

```
\section development_links Creating Links
```

20.3 Creating Links

Links to other documents and external links can be embedded with

- doxygen \ref links to other doxygen \page, \section, \subsection and \anchor locations
- HTML links without markup - doxygen creates "http://..." links automatically
- standard, non-Doxygen, HTML links

```
-   see chapter \ref unicode creates a link to the named chapter
    unicode that has been created with a \\page statement.

-   For further informations about quoting see
    http://www.doxygen.org/htmlcmds.html

-   see <a href="http://www.nedit.org/">Nedit</a> creates
    a standard HTML link
```

appears as:

- see chapter Unicode and UTF-8 Support creates a link to the named chapter unicode that has been created with a \page statement.
- For further informations about quoting see `http://www.doxygen.org/htmlcmds.html`
- see `Nedit` creates a standard HTML link

20.4 Paragraph Layout

There is no real need to use HTML <P> and </P> tags within the text to tell doxygen to start or stop a paragraph. In most cases, when doxygen encounters a blank line or some, but not all, **commands** in the text it knows that it has reached the start or end of a paragraph. Doxygen also offers the **par** command for special paragraph handling. It can be used to provide a paragraph title and also to indent a paragraph. Unfortunately **par** won't do what you expect if you want to have doxygen links and sometimes html tags don't work either.

```
\par Normal Paragraph with title

This paragraph will have a title, but because there is a blank line
between the \par and the text, it will have the normal layout.

\par Indented Paragraph with title
This paragraph will also have a title, but because there is no blank
line between the \par and the text, it will be indented.

\par
It is also possible to have an indented paragraph without title.
This is how you indent subsequent paragraphs.

\par No link to Fl_Widget::draw()
Note that the paragraph title is treated as plain text.
Doxygen type links will not work.
HTML characters and tags may or may not work.

Fl_Widget::draw() links and &quot;html&quot; tags work<br>
\par
Use a single line ending with <br> for complicated paragraph titles.
```

The above code produces the following paragraphs:

Normal Paragraph with title

This paragraph will have a title, but because there is a blank line between the \par and the text, it will have the normal layout.

Indented Paragraph with title

> This paragraph will also have a title, but because there is no blank line between the \par and the text, it will be indented.
>
> It is also possible to have an indented paragraph without title. This is how you indent subsequent paragraphs.

No link to Fl_Widget::draw()

> Note that the paragraph title is treated as plain text. Doxygen type links will not work. HTML characters and tags may or may not work.

Fl_Widget::draw() links and ”html” tags work

> Use a single line ending with <br> for complicated paragraph titles.

20.5 Navigation Elements

Each introduction (tutorial) page ends with navigation elements. These elements must only be included in the html documentation, therefore they must be separated with \htmlonly and \endhtmlonly.

The following code gives the navigation bar at the bottom of this page:

```
\htmlonly
<hr>
<table summary="navigation bar" width="100%" border="0">
<tr>
  <td width="45%" align="LEFT">
    <a class="el" href="migration_1_3.html">
    [Prev]
    Migrating Code from FLTK 1.1 to 1.3
    </a>
  </td>
  <td width="10%" align="CENTER">
    <a class="el" href="index.html">[Index]</a>
  </td>
  <td width="45%" align="RIGHT">
    <a class="el" href="license.html">
    Software License
    [Next]
    </a>
  </td>
</tr>
</table>
\endhtmlonly
```

Chapter 21

Software License

December 11, 2001

The FLTK library and included programs are provided under the terms of the GNU Library General Public License (LGPL) with the following exceptions:

1. Modifications to the FLTK configure script, config header file, and makefiles by themselves to support a specific platform do not constitute a modified or derivative work.

   The authors do request that such modifications be contributed to the FLTK project - send all contributions through the "Software Trouble Report" on the following page: `http://www.fltk.org/str.php`

2. Widgets that are subclassed from FLTK widgets do not constitute a derivative work.

3. Static linking of applications and widgets to the FLTK library does not constitute a derivative work and does not require the author to provide source code for the application or widget, use the shared FLTK libraries, or link their applications or widgets against a user-supplied version of FLTK.

   If you link the application or widget to a modified version of FLTK, then the changes to FLTK must be provided under the terms of the LGPL in sections 1, 2, and 4.

4. You do not have to provide a copy of the FLTK license with programs that are linked to the FLTK library, nor do you have to identify the FLTK license in your program or documentation as required by section 6 of the LGPL.

   However, programs must still identify their use of FLTK. The following example statement can be included in user documentation to satisfy this requirement:

   [program/widget] is based in part on the work of the FLTK project (`http://www.fltk.org`).

GNU LIBRARY GENERAL PUBLIC LICENSE

Version 2, June 1991
Copyright (C) 1991 Free Software Foundation, Inc.
59 Temple Place - Suite 330, Boston, MA 02111-1307, USA
Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.
[This is the first released version of the library GPL. It is numbered 2 because it goes with version 2 of the ordinary GPL.]

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users.

This license, the Library General Public License, applies to some specially designated Free Software Foundation software, and to any other libraries whose authors decide to use it. You can use it for your libraries, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library, or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link a program with the library, you must provide complete object files to the recipients so that they can relink them with the library, after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

Our method of protecting your rights has two steps: (1) copyright the library, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the library.

Also, for each distributor's protection, we want to make certain that everyone understands that there is no warranty for this free library. If the library is modified by someone else and passed on, we want its recipients to know that what they have is not the original version, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that companies distributing free software will individually obtain patent licenses, thus in effect transforming the program into proprietary software. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License, which was designed for utility programs. This license, the GNU Library General Public License, applies to certain designated libraries. This license is quite different from the ordinary one; be sure to read it in full, and don't assume that anything in it is the same as in the ordinary license.

The reason we have a separate public license for some libraries is that they blur the distinction we usually make between modifying or adding to a program and simply using it. Linking a program with a library, without changing the library, is in some sense simply using the library, and is analogous to running a utility program or application program. However, in a textual and legal sense, the linked executable is a combined work, a derivative of the original library, and the ordinary General Public License treats it as such.

Because of this blurred distinction, using the ordinary General Public License for libraries did not effectively promote software sharing, because most developers did not use the libraries. We concluded that weaker conditions might promote sharing better.

However, unrestricted linking of non-free programs would deprive the users of those programs of all benefit from the free status of the libraries themselves. This Library General Public License is intended to permit developers of non-free programs to use free libraries, while preserving your freedom as a user of such programs to change the free libraries that are incorporated in them. (We have not seen how to achieve this as regards changes in header files, but we have achieved it as regards changes in the actual functions of the Library.) The hope is that this will lead to faster development of free libraries.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the libary" and a "work that uses the library". The former contains code derived from the library, while the latter only works together with the library.

Note that it is possible for a library to be covered by the ordinary General Public License rather than by this special one.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any software library which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Library General Public License (also called ”this License”). Each licensee is addressed as ”you”.

A ”library” means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The ”Library”, below, refers to any such software library or work which has been distributed under these terms. A ”work based on the Library” means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term ”modification”.)

”Source code” for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library’s complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) The modified work must itself be a software library.

b) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.

c) You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.

d) If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

(For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a ”work that uses the Library”. Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

However, linking a ”work that uses the Library” with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a ”work that uses the library”. The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

When a ”work that uses the Library” uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also compile or link a ”work that uses the Library” with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer’s own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable ”work that uses the Library”, as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)

b) Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.

c) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.

d) Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.

b) Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted

Chapter 22

Example Source Code

The FLTK distribution contains over 60 sample applications written in, or ported to, FLTK.

If the FLTK archive you received does not contain either an 'examples' or 'test' directory, you can download the complete FLTK distribution from `http://fltk.org/software.php`.

Most of the example programs were created while testing a group of widgets. They are not meant to be great achievements in clean C++ programming, but merely a test platform to verify the functionality of the FLTK library.

Note that extra example programs are also available in an additional 'examples' directory, but these are **NOT** built automatically when you build FLTK, unlike those in the 'test' directory shown below.

22.1 Example Applications

adjuster	arc	ask	bitmap	blocks	boxtype
browser	button	buttons	checkers	clock	colbrowser
color_chooser	cube	CubeView	cursor	curve	demo
device	doublebuffer	editor	fast_slow	file_chooser	fluid
fonts	forms	fractals	fullscreen	gl_overlay	glpuzzle
hello	help	iconize	image	inactive	input
input_choice	keyboard	label	line_style	list_visuals	mandelbrot
menubar	message	minimum	navigation	output	overlay
pack	pixmap_-browser	pixmap	preferences	radio	resizebox
resize	scroll	shape	subwindow	sudoku	symbols
tabs	threads	tile	tiled_image	unittests	utf8
valuators					

22.1.1 adjuster

`adjuster` shows a nifty little widget for quickly setting values in a great range.

22.1.2 arc

The `arc` demo explains how to derive your own widget to generate some custom drawings. The sample drawings use the matrix based arc drawing for some fun effects.

22.1.3 ask

`ask` shows some of FLTK's standard dialog boxes. Click the correct answers or you may end up in a loop, or you may end up in a loop, or you... .

22.1.4 bitmap

This simple test shows the use of a single color bitmap as a label for a box widget. Bitmaps are stored in the X11 '.bmp' file format and can be part of the source code.

22.1.5 blocks

A wonderful and addictive game that shows the usage of FLTK timers, graphics, and how to implement sound on all platforms. `blocks` is also a good example for the Mac OS X specific bundle format.

22.1.6 boxtype

`boxtype` gives an overview of readily available boxes and frames in FLTK. More types can be added by the application programmer. When using themes, FLTK shuffles boxtypes around to give your program a new look.

22.1.7 browser

`browser` shows the capabilities of the Fl_Browser widget. Important features tested are loading of files, line formatting, and correct positioning of the browser data window.

22.1.8 button

The `button` test is a simple demo of push-buttons and callbacks.

22.1.9 buttons

`buttons` shows a sample of FLTK button types.

22.1.10 checkers

Written by Steve Poulsen in early 1979, `checkers` shows how to convert a VT100 text-terminal based program into a neat application with a graphical UI. Check out the code that drags the pieces, and how the pieces are drawn by layering. Then tell me how to beat the computer at Checkers.

22.1.11 clock

The `clock` demo shows two analog clocks. The innards of the Fl_Clock widget are pretty interesting, explaining the use of timeouts and matrix based drawing.

22.1.12 colbrowser

`colbrowser` runs only on X11 systems. It reads */usr/lib/X11/rgb.txt* to show the color representation of every text entry in the file. This is beautiful, but only moderately useful unless your UI is written in *Motif*.

22.1.13 color_chooser

The `color_chooser` gives a short demo of FLTK's palette based color chooser and of the RGB based color wheel.

22.1.14 cube

The `cube` demo shows the speed of OpenGL. It also tests the ability to render two OpenGL buffers into a single window, and shows OpenGL text.

22.1.15 CubeView

`CubeView` shows how to create a UI containing OpenGL with Fluid.

22.1.16 cursor

The `cursor` demo shows all mouse cursor shapes that come standard with FLTK. The *fgcolor* and *bgcolor* sliders work only on few systems (some version of Irix for example).

22.1.17 curve

`curve` draws a nice Bezier curve into a custom widget. The *points* option for splines is not supported on all platforms.

22.1.18 demo

This tool allows quick access to all programs in the `test` directory. `demo` is based on the visuals of the IrixGL demo program. The menu tree can be changed by editing `test/demo.menu`.

22.1.19 device

Exercises the Fl_Image_Surface, Fl_Copy_Surface, and Fl_Printer classes to draw to an Fl_Image object, copy graphical data to the clipboard, and for print support.

Note

The clipboard.cxx program of the 'examples' directory is a clipboard watching application that continuously displays the textual or graphical content of the system clipboard (a.k.a pasteboard on Mac OS X) exercising Fl::paste().

22.1.20 doublebuffer

The `doublebuffer` demo shows the difference between a single buffered window, which may flicker during a slow redraw, and a double buffered window, which never flickers, but uses twice the amount of RAM. Some modern OS's double buffer all windows automatically to allow transparency and shadows on the desktop. FLTK is smart enough to not tripple buffer a window in that case.

22.1.21 editor

FLTK has two very different text input widgets. Fl_Input and derived classes are rather light weight, however Fl_Text_Editor is a complete port of *nedit* (with permission). The `editor` test is almost a full application, showing custom syntax highlighting and dialog creation.

22.1.22 fast_slow

`fast_slow` shows how an application can use the Fl_Widget::when() setting to receive different kinds of callbacks.

22.1.23 file_chooser

The standard FLTK `file_chooser` is the result of many iterations, trying to find a middle ground between a complex browser and a fast light implementation.

22.1.24 fonts

`fonts` shows all available text fonts on the host system. If your machine still has some pixmap based fonts, the supported sizes will be shown in bold face. Only the first 256 fonts will be listed.

22.1.25 forms

`forms` is an XForms program with very few changes. Search for "fltk" to find all changes necessary to port to fltk. This demo shows the different boxtypes. Note that some boxtypes are not appropriate for some objects.

22.1.26 fractals

`fractals` shows how to mix OpenGL, Glut and FLTK code. FLTK supports a rather large subset of Glut, so that many Glut applications compile just fine.

22.1.27 fullscreen

This demo shows how to do many of the window manipulations that are popular for games. You can toggle the border on/off, switch between single- and double-buffered rendering, and take over the entire screen. More information in the source code.

22.1.28 gl_overlay

`gl_overlay` shows OpenGL overlay plane rendering. If no hardware overlay plane is available, FLTK will simulate it for you.

22.1.29 glpuzzle

The `glpuzzle` test shows how most Glut source code compiles easily under FLTK.

22.1.30 hello

`hello`: Hello, World. Need I say more? Well, maybe. This tiny demo shows how little is needed to get a functioning application running with FLTK. Quite impressive, I'd say.

22.1.31 help

`help` displays the built-in FLTK help browser. The Fl_Help_Dialog understands a subset of html and renders various image formats. This widget makes it easy to provide help pages to the user without depending on the operating system's html browser.

22.1.32 iconize

`iconize` demonstrates the effect of the window functions hide(), iconize(), and show().

22.1.33 image

The `image` demo shows how an image can be created on the fly. This generated image contains an alpha (transparency) channel which lets previous renderings 'shine through', either via true transparency or by using screen door transparency (pixelation).

22.1.34 inactive

`inactive` tests the correct rendering of inactive widgets. To see the inactive version of images, you can check out the pixmap or image test.

22.1.35 input

This tool shows and tests different types of text input fields based on Fl_Input_. The `input` program also tests various settings of Fl_Input::when().

22.1.36 input_choice

`input_choice` tests the latest addition to FLTK1, a text input field with an attached pulldown menu. Windows users will recognize similarities to the 'ComboBox'. `input_choice` starts up in 'plastic' scheme, but the traditional scheme is also supported.

22.1.37 keyboard

FLTK unifies keyboard events for all platforms. The `keyboard` test can be used to check the return values of Fl::event_key() and Fl::event_text(). It is also great to see the modifier buttons and the scroll wheel at work. Quit this application by closing the window. The ESC key will not work.

22.1.38 label

Every FLTK widget can have a label attached to it. The `label` demo shows alignment, clipping, and wrapping of text labels. Labels can contain symbols at the start and end of the text, like *@FLTK* or *@circle uh-huh @square*.

22.1.39 line_style

Advanced line drawing can be tested with `line_style`. Not all platforms support all line styles.

22.1.40 list_visuals

This little app finds all available pixel formats for the current X11 screen. But since you are now an FLTK user, you don't have to worry about any of this.

22.1.41 mandelbrot

`mandelbrot` shows two advanced topics in one test. It creates grayscale images on the fly, updating them via the *idle* callback system. This is one of the few occasions where the *idle* callback is very useful by giving all available processor time to the application without blocking the UI or other apps.

22.1.42 menubar

The `menubar` tests many aspects of FLTK's popup menu system. Among the features are radio buttons, menus taller than the screen, arbitrary sub menu depth, and global shortcuts.

22.1.43 message

`message` pops up a few of FLTK's standard message boxes.

22.1.44 minimum

The `minimum` test program verifies that the update regions are set correctly. In a real life application, the trail would be avoided by choosing a smaller label or by setting label clipping differently.

22.1.45 navigation

`navigation` demonstrates how the text cursor moves from text field to text field when using the arrow keys, tab, and shift-tab.

22.1.46 output

`output` shows the difference between the single line and multi line mode of the Fl_Output widget. Fonts can be selected from the FLTK standard list of fonts.

22.1.47 overlay

The `overlay` test app shows how easy an FLTK window can be layered to display cursor and manipulator style elements. This example derives a new class from Fl_Overlay_Window and provides a new function to draw custom overlays.

22.1.48 pack

The `pack` test program demonstrates the resizing and repositioning of children of the Fl_Pack group. Putting an Fl_Pack into an Fl_Scroll is a useful way to create a browser for large sets of data.

22.1.49 pixmap_browser

`pixmap_browser` tests the shared-image interface. When using the same image multiple times, Fl_Shared_Image will keep it only once in memory.

22.1.50 pixmap

This simple test shows the use of a LUT based pixmap as a label for a box widget. Pixmaps are stored in the X11 '.xpm' file format and can be part of the source code. Pixmaps support one transparent color.

22.1.51 preferences

I do have my `preferences` in the morning, but sometimes I just can't remember a thing. This is where the Fl_Preferences come in handy. They remember any kind of data between program launches.

22.1.52 radio

The `radio` tool was created entirely with *fluid*. It shows some of the available button types and tests radio button behavior.

22.1.53 resizebox

`resizebox` shows some possible ways of FLTK's automatic resize behavior.

22.1.54 resize

The `resize` demo tests size and position functions with the given window manager.

22.1.55 scroll

`scroll` shows how to scroll an area of widgets, one of them being a slow custom drawing. Fl_Scroll uses clipping and smart window area copying to improve redraw speed. The buttons at the bottom of the window control decoration rendering and updates.

22.1.56 shape

`shape` is a very minimal demo that shows how to create your own OpenGL rendering widget. Now that you know that, go ahead and write that flight simulator you always dreamt of.

22.1.57 subwindow

The `subwindow` demo tests messaging and drawing between the main window and 'true' sub windows. A sub window is different to a group by resetting the FLTK coordinate system to 0, 0 in the top left corner. On Win32 and X11, subwindows have their own operating system specific handle.

22.1.58 sudoku

Another highly addictive game - don't play it, I warned you. The implementation shows how to create application icons, how to deal with OS specifics, and how to generate sound.

22.1.59 symbols

`symbols` are a speciality of FLTK. These little vector drawings can be integrated into labels. They scale and rotate, and with a little patience, you can define your own. The rotation number refers to 45 degree rotations if you were looking at a numeric keypad (2 is down, 6 is right, etc.).

22.1.60 tabs

The `tabs` tool was created with *fluid*. It tests correct hiding and redisplaying of tabs, navigation across tabs, resize behavior, and no unneeded redrawing of invisible widgets.

The `tabs` application shows the Fl_Tabs widget on the left and the Fl_Wizard widget on the right side for direct comparison of these two panel management widgets.

22.1.61 threads

FLTK can be used in a multithreading environment. There are some limitations, mostly due to the underlying operating system. `threads` shows how to use Fl::lock(), Fl::unlock(), and Fl::awake() in secondary threads to keep FLTK happy. Although locking works on all platforms, this demo is not available on every machine.

22.1.62 tile

The `tile` tool shows a nice way of using Fl_Tile. To test correct resizing of subwindows, the widget for region 1 is created from an Fl_Window class.

22.1.63 tiled_image

The `tiled_image` demo uses an image as the background for a window by repeating it over the full size of the widget. The window is resizable and shows how the image gets repeated.

22.1.64 unittests

`unittests` exercises all of FLTK's drawing features (e.g., text, lines, circles, images), as well as scrollbars and schemes.

22.1.65 utf8

`utf8` shows all fonts available to the platform that runs it, and how each font draws each of the Unicode code points ranging between U+0020 and U+FFFF.

22.1.66 valuators

`valuators` shows all of FLTK's nifty widgets to change numeric values.

22.1.67 fluid

`fluid` is not only a big test program, but also a very useful visual UI designer. Many parts of `fluid` were created using `fluid`. See the Fluid Tutorial for more details.

Chapter 23

FAQ (Frequently Asked Questions)

A list of frequently asked questions about FLTK.

This appendix describes various frequently asked questions regarding FLTK.

23.1 Where do I start learning FLTK?

It is assumed you know C++, which is the language all FLTK programs are written in, including FLTK itself.

If you like reading manuals to work your way into things, a good start is the FLTK documentation's Introduction to FLTK. Under the FLTK Basics section there's an example 'hello world' program that includes a line-by-line description.

If you like looking at simple code first to pique your interest, and then read up from there, start with the example programs in the test/ and examples/ directory that is included with the source code. A good place to start is the 'hello world' program in test/hello.cxx. Also do a google search for "FLTK example programs". "Erco's Cheat Page" is one that shows many simple examples of how to do specific things.

If you like to run example programs and look for ones that are like yours and then read them, download and build FLTK from the source, then run the test/demo program. Also, go into the 'examples/' directory and run 'make', then run some of those programs.

If you prefer watching TV to reading books and code, google search for "FLTK video tutorials" which has some introductory examples of how to write FLTK programs in C++ and build them.

23.2 How do I make a box with text?

The 'hello world' program shows how to make a box with text. All widgets have labels, so picking a simple widget like Fl_Box and setting its label() and using align() to align the label and labelfont() to set the font, and labelsize() to set the size, you can get text just how you want.

Labels are not selectable though; if you want selectable text, you can use Fl_Output or Fl_Multiline_Output for simple text that doesn't include scrollbars. For more complex text that might want scrollbars and multiple colors/fonts, use either Fl_Text_Display which handles plain text, or Fl_Help_View which handles simple HTML formatted text.

23.3 Can I use FLTK to make closed-source commercial applications?

Yes. The FLTK Software License is standard LGPL, but also includes a special clause (”exception”) to allow for static linking. Specifically:

```
[from the top of the FLTK LGPL License section on exceptions]

3. Static linking of applications and widgets to the FLTK library does
not constitute a derivative work and does not require the author to
provide source code for the application or widget, use the shared FLTK
libraries, or link their applications or widgets against a user-supplied
version of FLTK.

If you link the application or widget to a modified version of FLTK,
then the changes to FLTK must be provided under the terms of the LGPL
in sections 1, 2, and 4.

4. You do not have to provide a copy of the FLTK license with programs
that are linked to the FLTK library, nor do you have to identify the
FLTK license in your program or documentation as required by section 6
of the LGPL.

However, programs must still identify their use of FLTK.  The following
example statement can be included in user documentation to satisfy
this requirement:

    [program/widget] is based in part on the work of the
    FLTK project (http://www.fltk.org).
```

23.4 Hitting the ’Escape’ key closes windows - how do I prevent this?

[From FLTK article #378]

1. FLTK has a ”global event handler” that makes Escape try to close the window, the same as clicking the close box. To disable this everywhere you can install your own that pretends it wants the escape key and thus stops the default one from seeing it (this may not be what you want, see below about the callbacks):

```
static int my_handler(int event) {
  if (event == FL_SHORTCUT) return 1; // eat all shortcut keys
  return 0;
}
...in main():
  Fl::add_handler(my_handler);
...
```

1. Attempts to close a window (both clicking the close box or typing Escape) call that window’s callback. The default version of the callback does hide(). To make the window not close or otherwise do something different you replace the callback. To make the main window exit the program:

```
void my_callback(Fl_Widget*, void*) {
  exit(0);
}
...
  main_window->callback(my_callback);
...
```

If you don’t want Escape to close the main window and exit you can check for and ignore it. This is better than replacing the global handler because Escape will still close pop-up windows:

```
void my_callback(Fl_Widget*, void*) {
  if (Fl::event()==FL_SHORTCUT && Fl::event_key()==
      FL_Escape)
    return; // ignore Escape
  exit(0);
}
```

It is very common to ask for confirmation before exiting, this can be done with:

```
void my_callback(Fl_Widget*, void*) {
  if (fl_ask("Are you sure you want to quit?"))
    exit(0);
}
```

Chapter 24

Todo List

Page Adding and Extending Widgets

Clarify Fl_Window::damage(uchar) handling - seems confused/wrong? ORing value doesn't match setting behaviour in FL_Widget.H!

Clarify Fl_Widget::test_shortcut() explanations. Fl_Widget.h says Internal Use only, but subclassing chapter gives details!

Group Box Types

Description of boxtypes is incomplete. See below for the defined enum Fl_Boxtype.

See Also

src/Fl_get_system_colors.cxx

Page Drawing Things in FLTK

add an Fl_Draw_Area_Cb typedef to allow fl_scroll(...) to be doxygenated?

Member Fl_Browser_::scrollbar_width () const

This method should eventually be removed in 1.4+

Member Fl_Browser_::scrollbar_width (int width)

This method should eventually be removed in 1.4+

Member Fl_Browser_::sort (int flags=0)

Add a flag to ignore case

Class Fl_Button

Refactor the doxygen comments for Fl_Button type() documentation.

Refactor the doxygen comments for Fl_Button when() documentation.

Class Fl_Chart

Refactor Fl_Chart::type() information.

Class Fl_Choice

Refactor the doxygen comments for Fl_Choice changed() documentation.

Class Fl_Counter

Refactor the doxygen comments for Fl_Counter type() documentation.

Member Fl_Cursor

enum Fl_Cursor needs maybe an image.

Member Fl_File_Input::errorcolor () const

Better docs for Fl_File_Input::errorcolor() - is it even used?

Member Fl_Group::sizes ()

Should the internal representation of the sizes() array be documented?

Member fl_height (int font, int size)

In the future, when the XFT issues are resolved, this function should simply return the 'size' value.

Member Fl_Input_::handle_mouse (int, int, int, int, int keepmark=0)

Add comment and parameters

Member Fl_Input_::handletext (int e, int, int, int, int)

Add comment and parameters

Member fl_intptr_t

typedef's fl_intptr_t and fl_uintptr_t should be documented.

Class Fl_Label

There is an aspiration that the Fl_Label type will become a widget by itself. That way we will be avoiding a lot of code duplication by handling labels in a similar fashion to widgets containing text. We also provide an easy interface for very complex labels, containing html or vector graphics. However, this re-factoring is not in place in this release.

Member Fl_Labeltype

The doxygen comments are incomplete, and some labeltypes start with an underscore. Also, there are three external functions undocumented (yet):

- fl_define_FL_SHADOW_LABEL()
- fl_define_FL_ENGRAVED_LABEL()
- fl_define_FL_EMBOSSED_LABEL()

Member Fl_Menu_::add (const char ∗, int shortcut, Fl_Callback ∗, void ∗=0, int=0)

Raw integer shortcut needs examples. Dependent on responses to `http://fltk.org/newsgroups.-php?gfltk.development+v:10086` and results of STR#2344

Member fl_old_shortcut (const char ∗s)

Fix these silly legacy issues in a future release to support more predictable behavior for the modifier keys.

Member Fl_Preferences::get (const char ∗entry, void ∗value, const void ∗defaultValue, int defaultSize, int maxSize)

maxSize should receive the number of bytes that were read.

Member fl_reset_spot (void)

provide user documentation for fl_reset_spot function

Member Fl_Scroll::bbox (int &, int &, int &, int &)

The visibility of the scrollbars ought to be checked/calculated outside of the draw() method (STR #1895).

Member fl_set_spot (int font, int size, int X, int Y, int W, int H, Fl_Window ∗win=0)

provide user documentation for fl_set_spot function

Member fl_set_status (int X, int Y, int W, int H)

provide user documentation for fl_set_status function

Member Fl_String

FIXME: temporary (?) typedef to mark UTF-8 and Unicode conversions

Member Fl_Text_Display::display_insert ()

Unicode?

Member Fl_Text_Display::extend_range_for_styles (int ∗start, int ∗end)

Unicode?

Member Fl_Text_Display::handle_vline (int mode, int lineStart, int lineLen, int leftChar, int rightChar, int topClip, int bottomClip, int leftClip, int rightClip) const

we need to handle hidden hyphens and tabs here!

we handle all styles and selections

we must provide code to get pixel positions of the middle of a character as well

Member Fl_Text_Display::overstrike (const char ∗text)

Unicode? Find out exactly what we do here and simplify.

Member Fl_Text_Display::position_to_line (int pos, int ∗lineNum) const

What does this do?

Member Fl_Text_Display::position_to_linecol (int pos, int ∗lineNum, int ∗column) const

a column number makes little sense in the UTF-8/variable font width environment. We will have to further define what exactly we want to return. Please check the functions that call this particular function.

Member Fl_Text_Display::scroll (int topLineNum, int horizOffset)

Column numbers make little sense here.

Member Fl_Text_Display::shortcut () const

FIXME : get set methods pointing on shortcut_ have no effects as shortcut_ is unused in this class and derived!

Returns

the current shortcut key

Member Fl_Text_Display::shortcut (int s)

FIXME : get set methods pointing on shortcut_ have no effects as shortcut_ is unused in this class and derived!

Parameters

s	the new shortcut key

Member Fl_Text_Display::wrap_mode (int wrap, int wrap_margin)

we need new wrap modes to wrap at the window edge and based on pixel width or average character width.

Member Fl_Text_Display::wrapped_column (int row, int column) const

What does this do and how is it useful? Column numbers mean little in this context. Which functions depend on this one?

Unicode?

Member Fl_Text_Display::wrapped_row (int row) const

What does this do and how is it useful? Column numbers mean little in this context. Which functions depend on this one?

Member Fl_Tiled_Image::Fl_Tiled_Image (Fl_Image ∗i, int W=0, int H=0)

Fix Fl_Tiled_Image as background image for widgets and windows and fix the implementation of Fl::scheme(const char ∗).

Member Fl_Tree::handle (int e)

add Fl_Widget_Tracker (see Fl_Browser_.cxx::handle())

Member Fl_Tree::is_scrollbar (Fl_Widget ∗w)

should be const

Member Fl_Tree::show_self ()

should be const

Version

1.3.0

Member Fl_When

doxygen comments for values are incomplete and maybe wrong or unclear

Member Fl_Widget::argument (long v)

The user data value must be implemented using *intptr_t* or similar to avoid 64-bit machine incompatibilities.

Member Fl_Widget::argument () const

The user data value must be implemented using *intptr_t* or similar to avoid 64-bit machine incompatibilities.

Member Fl_Widget::type () const

Explain "simulate RTTI" (currently only used to decide if a widget is a window, i.e. type()>=FL_WINDOW ?). Is type() really used in a way that ensures "Forms compatibility" ?

Member Fl_Window::show (int argc, char ∗∗argv)

explain which system parameters are set up.

Member Fl_Window::show ()

Check if we can remove resetting the current group in a later FLTK version (after 1.3.x). This may break "already broken" programs though if they rely on this "feature".

Page Handling Events

Add details on how to detect repeating keys, since on some X servers a repeating key will generate both FL_KEYUP and FL_KEYDOWN, such that to tell if a key is held, you need Fl::event_key(int) to detect if the key is being held down during FL_KEYUP or not.

Group Mouse and Keyboard Events

FL_Button and FL_key... constants could be structured better (use an enum or some doxygen grouping ?)

Page Unicode and UTF-8 Support

Do we need this info about planes?

Work through the code and this documentation to harmonize the [**OksiD**] and [**fltk2**] functions.

Verify 16/24 bit Unicode limit for different character sets? OksiD's code appears limited to 16-bit whereas the FLTK2 code appears to handle a wider set. What about illegal characters? See comments in fl_utf8fromwc() and fl_utf8toUtf16().

Chapter 25

Deprecated List

Member Fl::release ()

Use Fl::grab(0) instead.

See Also

grab(Fl_Window*)

Member Fl::set_idle (Fl_Old_Idle_Handler cb)

This method is obsolete - use the add_idle() method instead.

Member Fl::version ()

Use int Fl::api_version() instead.

Member fl_ask (const char *fmt,...)

fl_ask() is deprecated since it uses "Yes" and "No" for the buttons which does not conform to the current FLTK Human Interface Guidelines. Use fl_choice() with the appropriate verbs instead.

Member fl_clip

fl_clip(int, int, int, int) is deprecated and will be removed from future releases. Please use fl_push_-clip(int x, int y, int w, int h) instead.

Member Fl_Group::focus (Fl_Widget *W)

This is for backwards compatibility only. You should use *W->take_focus*() instead.

See Also

Fl_Widget::take_focus();

Member Fl_Menu_Item::check ()

.

Member Fl_Menu_Item::checked () const

.

Member Fl_Menu_Item::setonly ()

This method is dangerous if radio items are first in the menu. Use Fl_Menu_::setonly(Fl_Menu_Item*) instead.

Member Fl_Menu_Item::uncheck ()

.

Member Fl_Spinner::maxinum () const

Member Fl_Spinner::mininum () const

Member Fl_Tree::first_visible ()

in 1.3.3 ABI – use first_visible_item() instead.

Member Fl_Tree::item_clicked ()

in 1.3.3 ABI – use callback_item() instead.

Member Fl_Tree::item_clicked (Fl_Tree_Item ∗val)

in 1.3.3 ABI – use callback_item() instead.

Member Fl_Tree::last_visible ()

in 1.3.3 – use last_visible_item() instead.

Member Fl_Tree_Item::Fl_Tree_Item (const Fl_Tree_Prefs &prefs)

in 1.3.3 ABI – you must use Fl_Tree_Item(Fl_Tree∗) for proper horizontal scrollbar behavior.

Member Fl_Tree_Item::next_displayed (Fl_Tree_Prefs &prefs)

in 1.3.3 for confusing name, use next_visible() instead

Member Fl_Tree_Item::prev_displayed (Fl_Tree_Prefs &prefs)

in 1.3.3 for confusing name, use prev_visible()

Member FL_VERSION

This `double` version number is retained for compatibility with existing program code. New code should use *int* FL_API_VERSION instead. FL_VERSION is deprecated because comparisons of floating point values may fail due to rounding errors. However, there are currently no plans to remove this deprecated constant.

Member Fl_Widget::color2 (unsigned a)

Use selection_color(unsigned) instead.

Member Fl_Widget::color2 () const

Use selection_color() instead.

Member Fl_Window::free_position ()

please use force_position(0) instead

Member Fl_Window::icon (const void ∗ic)

in 1.3.3

Member Fl_Window::icon () const

in 1.3.3

Chapter 26

Module Index

26.1 Modules

Here is a list of all modules:

Chapter 27

Hierarchical Index

27.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 28

Class Index

28.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 29

File Index

29.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 30

Module Documentation

30.1 Callback function typedefs

Typedefs defined in <FL/Fl.H> for callback or handler functions passed as function parameters.

Typedefs

- typedef void(∗ Fl_Abort_Handler)(const char ∗format,...)

  Signature of set_abort functions passed as parameters.
- typedef int(∗ Fl_Args_Handler)(int argc, char ∗∗argv, int &i)

  Signature of args functions passed as parameters.
- typedef void(∗ Fl_Atclose_Handler)(Fl_Window ∗window, void ∗data)

  Signature of set_atclose functions passed as parameters.
- typedef void(∗ Fl_Awake_Handler)(void ∗data)

  Signature of some wakeup callback functions passed as parameters.
- typedef void(Fl_Box_Draw_F)(int x, int y, int w, int h, Fl_Color color)

  Signature of some box drawing functions passed as parameters.
- typedef void(∗ Fl_Clipboard_Notify_Handler)(int source, void ∗data)

  Signature of add_clipboard_notify functions passed as parameters.
- typedef int(∗ Fl_Event_Dispatch)(int event, Fl_Window ∗w)

  Signature of event_dispatch functions passed as parameters.
- typedef int(∗ Fl_Event_Handler)(int event)

  Signature of add_handler functions passed as parameters.
- typedef void(∗ Fl_FD_Handler)(FL_SOCKET fd, void ∗data)

  Signature of add_fd functions passed as parameters.
- typedef void(∗ Fl_Idle_Handler)(void ∗data)

  Signature of add_idle callback functions passed as parameters.
- typedef void(Fl_Label_Draw_F)(const Fl_Label ∗label, int x, int y, int w, int h, Fl_Align align)

  Signature of some label drawing functions passed as parameters.
- typedef void(Fl_Label_Measure_F)(const Fl_Label ∗label, int &width, int &height)

  Signature of some label measurement functions passed as parameters.
- typedef void(∗ Fl_Old_Idle_Handler)()

  Signature of set_idle callback functions passed as parameters.
- typedef int(∗ Fl_System_Handler)(void ∗event, void ∗data)

  Signature of add_system_handler functions passed as parameters.
- typedef void(∗ Fl_Timeout_Handler)(void ∗data)

  Signature of some timeout callback functions passed as parameters.

30.1.1 Detailed Description

Typedefs defined in <FL/Fl.H> for callback or handler functions passed as function parameters. FLTK uses callback functions as parameters for some function calls, e.g. to set up global event handlers (Fl::add_handler()), to add a timeout handler (Fl::add_timeout()), and many more.

The typedefs defined in this group describe the function parameters used to set up or clear the callback functions and should also be referenced to define the callback function to handle such events in the user's code.

See Also

Fl::add_handler(), Fl::add_timeout(), Fl::repeat_timeout(), Fl::remove_timeout() and others

30.1.2 Typedef Documentation

typedef int(∗ Fl_Event_Dispatch)(int event, Fl_Window ∗w)

Signature of event_dispatch functions passed as parameters.

See Also

Fl::event_dispatch(Fl_Event_Dispatch)

30.2 Windows handling functions

Windows and standard dialogs handling declared in <FL/Fl.H>

Functions

- static void Fl::default_atclose (Fl_Window ∗, void ∗)

    Default callback for window widgets.
- static Fl_Window ∗ Fl::first_window ()

    Returns the first top-level window in the list of shown() windows.
- static void Fl::first_window (Fl_Window ∗)

    Sets the window that is returned by first_window().
- static Fl_Window ∗ Fl::grab ()

    Returns the window that currently receives all events.
- static void Fl::grab (Fl_Window ∗)

    Selects the window to grab.
- static Fl_Window ∗ Fl::modal ()

    Returns the top-most modal() window currently shown.
- static Fl_Window ∗ Fl::next_window (const Fl_Window ∗)

    Returns the next top-level window in the list of shown() windows.
- static void Fl::set_abort (Fl_Abort_Handler f)

    For back compatibility, sets the void Fl::fatal handler callback.
- static void Fl::set_atclose (Fl_Atclose_Handler f)

    For back compatibility, sets the Fl::atclose handler callback.

Variables

- static void(∗ Fl::atclose)(Fl_Window ∗, void ∗)

    Back compatibility: default window callback handler.

30.2.1 Detailed Description

Windows and standard dialogs handling declared in <FL/Fl.H>

30.2.2 Function Documentation

void Fl::default_atclose (Fl_Window ∗ *window*, void ∗ *v*) `[static]`

Default callback for window widgets.

It hides the window and then calls the default widget callback.

Fl_Window ∗ Fl::first_window ( ) `[static]`

Returns the first top-level window in the list of shown() windows.

If a modal() window is shown this is the top-most modal window, otherwise it is the most recent window to get an event.

void Fl::first_window (Fl_Window ∗ *window*) `[static]`

Sets the window that is returned by first_window().

The window is removed from wherever it is in the list and inserted at the top. This is not done if Fl::modal() is on or if the window is not shown(). Because the first window is used to set the ”parent” of modal windows, this is often useful.

static Fl_Window∗ Fl::grab ( ) `[inline], [static]`

Returns the window that currently receives all events.

Returns

The window that currently receives all events, or NULL if event grabbing is currently OFF.

void Fl::grab (Fl_Window ∗ *win*) `[static]`

Selects the window to grab.

This is used when pop-up menu systems are active.

Send all events to the passed window no matter where the pointer or focus is (including in other programs). The window *does not have to be shown()* , this lets the handle() method of a "dummy" window override all event handling and allows you to map and unmap a complex set of windows (under both X and WIN32 *some* window must be mapped because the system interface needs a window id).

If grab() is on it will also affect show() of windows by doing system-specific operations (on X it turns on override-redirect). These are designed to make menus popup reliably and faster on the system.

To turn off grabbing do Fl::grab(0).

Be careful that your program does not enter an infinite loop while grab() is on. On X this will lock up your screen! To avoid this potential lockup, all newer operating systems seem to limit mouse pointer grabbing to the time during which a mouse button is held down. Some OS's may not support grabbing at all.

static Fl_Window∗ Fl::modal ( ) `[inline], [static]`

Returns the top-most modal() window currently shown.

This is the most recently shown() window with modal() true, or NULL if there are no modal() windows shown(). The modal() window has its handle() method called for all events, and no other windows will have handle() called (grab() overrides this).

Fl_Window ∗ Fl::next_window (const Fl_Window ∗ *window*) `[static]`

Returns the next top-level window in the list of shown() windows.

You can use this call to iterate through all the windows that are shown().

Parameters

`in`	*window*	must be shown and not NULL

static void Fl::set_atclose (Fl_Atclose_Handler *f*) `[inline], [static]`

For back compatibility, sets the Fl::atclose handler callback.

You can now simply change the callback for the window instead.

See Also

Fl_Window::callback(Fl_Callback∗)

30.2.3 Variable Documentation

void(∗ Fl::atclose)(Fl_Window ∗, void ∗)=default_atclose `[static], [default]`

Back compatibility: default window callback handler.

See Also

Fl::set_atclose()

30.3 Events handling functions

Fl class events handling API declared in <FL/Fl.H>

Functions

- static void Fl::add_handler (Fl_Event_Handler h)

  Install a function to parse unrecognized events.
- static void Fl::add_system_handler (Fl_System_Handler h, void ∗data)

  Install a function to intercept system events.
- static Fl_Widget ∗ Fl::belowmouse ()

  Gets the widget that is below the mouse.
- static void Fl::belowmouse (Fl_Widget ∗)

  Sets the widget that is below the mouse.
- static int Fl::compose (int &del)

  Any text editing widget should call this for each FL_KEYBOARD event.
- static void Fl::compose_reset ()

  If the user moves the cursor, be sure to call Fl::compose_reset().
- static void Fl::disable_im ()

  Disables the system input methods facilities.
- static void Fl::enable_im ()

  Enables the system input methods facilities.
- static int Fl::event ()

  Returns the last event that was processed.
- static int Fl::event_alt ()

  Returns non-zero if the Alt key is pressed.
- static int Fl::event_button ()

  Gets which particular mouse button caused the current event.
- static int Fl::event_button1 ()

  Returns non-zero if mouse button 1 is currently held down.
- static int Fl::event_button2 ()

  Returns non-zero if button 2 is currently held down.
- static int Fl::event_button3 ()

  Returns non-zero if button 3 is currently held down.
- static int Fl::event_buttons ()

  Returns the mouse buttons state bits; if non-zero, then at least one button is pressed now.
- static int Fl::event_clicks ()

  Returns non zero if we had a double click event.
- static void Fl::event_clicks (int i)

  Manually sets the number returned by Fl::event_clicks().
- static void ∗ Fl::event_clipboard ()

  During an FL_PASTE event of non-textual data, returns a pointer to the pasted data.
- static const char ∗ Fl::event_clipboard_type ()

  Returns the type of the pasted data during an FL_PASTE event.
- static int Fl::event_command ()

  Returns non-zero if the FL_COMMAND key is pressed, either FL_CTRL or on OSX FL_META.
- static int Fl::event_ctrl ()

Returns non-zero if the Control key is pressed.

- static void Fl::event_dispatch (Fl_Event_Dispatch d)

  Set a new event dispatch function.

- static Fl_Event_Dispatch Fl::event_dispatch ()

  Return the current event dispatch function.

- static int Fl::event_dx ()

  Returns the current horizontal mouse scrolling associated with the FL_MOUSEWHEEL event.

- static int Fl::event_dy ()

  Returns the current vertical mouse scrolling associated with the FL_MOUSEWHEEL event.

- static int Fl::event_inside (int, int, int, int)

  Returns whether or not the mouse event is inside the given rectangle.

- static int Fl::event_inside (const Fl_Widget ∗)

  Returns whether or not the mouse event is inside a given child widget.

- static int Fl::event_is_click ()

  Returns non-zero if the mouse has not moved far enough and not enough time has passed since the last FL_PUSH or FL_KEYBOARD event for it to be considered a "drag" rather than a "click".

- static void Fl::event_is_click (int i)

  Clears the value returned by Fl::event_is_click().

- static int Fl::event_key ()

  Gets which key on the keyboard was last pushed.

- static int Fl::event_key (int key)

  Returns true if the given `key` was held down (or pressed) during the last event.

- static int Fl::event_length ()

  Returns the length of the text in Fl::event_text().

- static int Fl::event_original_key ()

  Returns the keycode of the last key event, regardless of the NumLock state.

- static int Fl::event_shift ()

  Returns non-zero if the Shift key is pressed.

- static int Fl::event_state ()

  Returns the keyboard and mouse button states of the last event.

- static int Fl::event_state (int mask)

  Returns non-zero if any of the passed event state bits are turned on.

- static const char ∗ Fl::event_text ()

  Returns the text associated with the current event, including FL_PASTE or FL_DND_RELEASE events.

- static int Fl::event_x ()

  Returns the mouse position of the event relative to the Fl_Window it was passed to.

- static int Fl::event_x_root ()

  Returns the mouse position on the screen of the event.

- static int Fl::event_y ()

  Returns the mouse position of the event relative to the Fl_Window it was passed to.

- static int Fl::event_y_root ()

  Returns the mouse position on the screen of the event.

- static Fl_Widget ∗ Fl::focus ()

  Gets the current Fl::focus() widget.

- static void Fl::focus (Fl_Widget ∗)

  Sets the widget that will receive FL_KEYBOARD events.

- static int Fl::get_key (int key)

Returns true if the given `key` is held down now.
- static void Fl::get_mouse (int &, int &)

    Return where the mouse is on the screen by doing a round-trip query to the server.
- static int Fl::handle (int, Fl_Window ∗)

    Handle events from the window system.
- static int Fl::handle_ (int, Fl_Window ∗)

    Handle events from the window system.
- static Fl_Widget ∗ Fl::pushed ()

    Gets the widget that is being pushed.
- static void Fl::pushed (Fl_Widget ∗)

    Sets the widget that is being pushed.
- static void Fl::remove_handler (Fl_Event_Handler h)

    Removes a previously added event handler.
- static void Fl::remove_system_handler (Fl_System_Handler h)

    Removes a previously added system event handler.
- static int Fl::test_shortcut (Fl_Shortcut)

    Tests the current event, which must be an FL_KEYBOARD or FL_SHORTCUT, against a shortcut value (described in Fl_Button).

Variables

- const char ∗const fl_eventnames [ ]

    This is an array of event names you can use to convert event numbers into names.
- const char ∗const fl_fontnames [ ]

    This is an array of font names you can use to convert font numbers into names.

30.3.1 Detailed Description

Fl class events handling API declared in <FL/Fl.H>

30.3.2 Function Documentation

void Fl::add_handler (Fl_Event_Handler *ha*) **`[static]`**

Install a function to parse unrecognized events.

If FLTK cannot figure out what to do with an event, it calls each of these functions (most recent first) until one of them returns non-zero. If none of them returns non-zero then the event is ignored. Events that cause this to be called are:

- FL_SHORTCUT events that are not recognized by any widget. This lets you provide global shortcut keys.
- FL_SCREEN_CONFIGURATION_CHANGED events. Under X11, this event requires the libXrandr.-so shared library to be loadable at run-time and the X server to implement the RandR extension.
- FL_FULLSCREEN events sent to a window that enters or leaves fullscreen mode.
- System events that FLTK does not recognize. See fl_xevent.
- *Some* other events when the widget FLTK selected returns zero from its handle() method. Exactly which ones may change in future versions, however.

See Also

Fl::remove_handler(Fl_Event_Handler)
Fl::event_dispatch(Fl_Event_Dispatch d)
Fl::handle(int, Fl_Window*)

void Fl::add_system_handler (Fl_System_Handler *ha*, void * *data*) `[static]`

Install a function to intercept system events.

FLTK calls each of these functions as soon as a new system event is received. The processing will stop at the first function to return non-zero. If all functions return zero then the event is passed on for normal handling by FLTK.

Each function will be called with a pointer to the system event as the first argument and `data` as the second argument. The system event pointer will always be void *, but will point to different objects depending on the platform:

- X11: XEvent
- Windows: MSG
- OS X: NSEvent

Parameters

ha	The event handler function to register
data	User data to include on each call

See Also

Fl::remove_system_handler(Fl_System_Handler)

static Fl_Widget* Fl::belowmouse ( ) `[inline]`, `[static]`

Gets the widget that is below the mouse.

See Also

belowmouse(Fl_Widget*)

void Fl::belowmouse (Fl_Widget * *o*) `[static]`

Sets the widget that is below the mouse.

This is for highlighting buttons. It is not used to send FL_PUSH or FL_MOVE directly, for several obscure reasons, but those events typically go to this widget. This is also the first widget tried for FL_SHORTCUT events.

If you change the belowmouse widget, the previous one and all parents (that don't contain the new widget) are sent FL_LEAVE events. Changing this does *not* send FL_ENTER to this or any widget, because sending FL_ENTER is supposed to *test* if the widget wants the mouse (by it returning non-zero from handle()).

int Fl::compose (int & *del*) `[static]`

Any text editing widget should call this for each FL_KEYBOARD event.

Use of this function is very simple.

If *true* is returned, then it has modified the Fl::event_text() and Fl::event_length() to a set of *bytes* to insert (it may be of zero length!). It will also set the "del" parameter to the number of *bytes* to the left of the cursor to delete, this is used to delete the results of the previous call to Fl::compose().

If *false* is returned, the keys should be treated as function keys, and del is set to zero. You could insert the text anyways, if you don't know what else to do.

On the Mac OS platform, text input can involve marked text, that is, temporary text replaced by other text during the input process. This occurs, e.g., when using dead keys or when entering CJK characters. Text editing widgets should preferentially signal marked text, usually underlining it. Widgets can use `int Fl::compose_state` *after* having called Fl::compose(int&) to obtain the length in bytes of marked text that always finishes at the current insertion point. It's the widget's task to underline marked text. Widgets should also call `void Fl::reset_marked_text()` when processing FL_UNFOCUS events. Optionally, widgets can also call `void Fl::insertion_point_location(int x, int y, int height)` to indicate the window coordinates of the bottom of the current insertion point and the line height. This way, auxiliary windows that help choosing among alternative characters appear just below the insertion point. If widgets don't do that, auxiliary windows appear at the widget's bottom. The Fl_Input and Fl_Text_Editor widgets underline marked text. If none of this is done by a user-defined text editing widget, text input will work, but will not signal to the user what text is marked. Finally, text editing widgets should call `set_flag(MAC_USE_ACCENTS_MENU);` in their constructor if they want to use the feature introduced with Mac OS 10.7 "Lion" where pressing and holding a key on the keyboard opens an accented-character menu window.

Though the current implementation returns immediately, future versions may take quite awhile, as they may pop up a window or do other user-interface things to allow characters to be selected.

void Fl::compose_reset ( ) `[static]`

If the user moves the cursor, be sure to call Fl::compose_reset().

The next call to Fl::compose() will start out in an initial state. In particular it will not set "del" to non-zero. This call is very fast so it is ok to call it many times and in many places.

static void Fl::disable_im ( ) `[static]`

Disables the system input methods facilities.

See Also

enable_im()

static void Fl::enable_im ( ) `[static]`

Enables the system input methods facilities.

This is the default.

See Also

disable_im()

static int Fl::event ( ) `[inline], [static]`

Returns the last event that was processed.

This can be used to determine if a callback is being done in response to a keypress, mouse click, etc.

static int Fl::event_alt ( ) `[inline], [static]`

Returns non-zero if the Alt key is pressed.

static int Fl::event_button ( ) `[inline], [static]`

Gets which particular mouse button caused the current event.

This returns garbage if the most recent event was not a FL_PUSH or FL_RELEASE event.

Return values

FL_LEFT_MOUSE	
FL_MIDDLE_MOUSE	
FL_RIGHT_MOUSE.	

See Also

Fl::event_buttons()

static int Fl::event_button1 ( ) `[inline], [static]`

Returns non-zero if mouse button 1 is currently held down.
For more details, see Fl::event_buttons().

static int Fl::event_button2 ( ) `[inline], [static]`

Returns non-zero if button 2 is currently held down.
For more details, see Fl::event_buttons().

static int Fl::event_button3 ( ) `[inline], [static]`

Returns non-zero if button 3 is currently held down.
For more details, see Fl::event_buttons().

static int Fl::event_buttons ( ) `[inline], [static]`

Returns the mouse buttons state bits; if non-zero, then at least one button is pressed now.
This function returns the button state at the time of the event. During an FL_RELEASE event, the state of the released button will be 0. To find out, which button caused an FL_RELEASE event, you can use Fl::event_button() instead.

Returns

a bit mask value like { [FL_BUTTON1] | [FL_BUTTON2] | [FL_BUTTON3] }

static int Fl::event_clicks ( ) `[inline], [static]`

Returns non zero if we had a double click event.
Return values

Non-zero	if the most recent FL_PUSH or FL_KEYBOARD was a "double click".
N-1	for N clicks. A double click is counted if the same button is pressed again while event_is_click() is true.

static void Fl::event_clicks (int *i*) `[inline], [static]`

Manually sets the number returned by Fl::event_clicks().
This can be used to set it to zero so that later code does not think an item was double-clicked.
Parameters

in	*i*	corresponds to no double-click if 0, i+1 mouse clicks otherwise

See Also

int event_clicks()

static void∗ Fl::event_clipboard ( ) [inline], [static]

During an FL_PASTE event of non-textual data, returns a pointer to the pasted data.

The returned data is an Fl_Image ∗ when the result of Fl::event_clipboard_type() is Fl::clipboard_image.

static const char∗ Fl::event_clipboard_type ( ) [inline], [static]

Returns the type of the pasted data during an FL_PASTE event.

This type can be Fl::clipboard_plain_text or Fl::clipboard_image.

static int Fl::event_command ( ) [inline], [static]

Returns non-zero if the FL_COMMAND key is pressed, either FL_CTRL or on OSX FL_META.

static int Fl::event_ctrl ( ) [inline], [static]

Returns non-zero if the Control key is pressed.

void Fl::event_dispatch (Fl_Event_Dispatch *d*) [static]

Set a new event dispatch function.

The event dispatch function is called after native events are converted to FLTK events, but before they are handled by FLTK. If the dispatch function Fl_Event_Dispatch `d` is set, it is up to the dispatch function to call Fl::handle_(int, Fl_Window∗) or to ignore the event.

The dispatch function itself must return 0 if it ignored the event, or non-zero if it used the event. If you call Fl::handle_(), then this will return the correct value.

The event dispatch can be used to handle exceptions in FLTK events and callbacks before they reach the native event handler:

```
int myHandler(int e, Fl_Window *w) {
  try {
    return Fl::handle_(e, w);
  } catch () {
    ...
  }
}

main() {
  Fl::event_dispatch(myHandler);
  ...
  Fl::run();
}
```

Parameters

d	new dispatch function, or NULL

See Also

Fl::add_handler(Fl_Event_Handler)
Fl::handle(int, Fl_Window∗)
Fl::handle_(int, Fl_Window∗)

static int Fl::event_dx ( ) `[inline], [static]`

Returns the current horizontal mouse scrolling associated with the FL_MOUSEWHEEL event.
Right is positive.

static int Fl::event_dy ( ) `[inline], [static]`

Returns the current vertical mouse scrolling associated with the FL_MOUSEWHEEL event.
Down is positive.

int Fl::event_inside (int *xx*, int *yy*, int *ww*, int *hh*) `[static]`

Returns whether or not the mouse event is inside the given rectangle.

Returns non-zero if the current Fl::event_x() and Fl::event_y() put it inside the given arbitrary bounding box.

You should always call this rather than doing your own comparison so you are consistent about edge effects.

To find out, whether the event is inside a child widget of the current window, you can use Fl::event_-inside(const Fl_Widget ∗).

Parameters

`in`	*xx,yy,ww,hh*	bounding box

Returns

non-zero, if mouse event is inside

int Fl::event_inside (const Fl_Widget ∗ *o*) `[static]`

Returns whether or not the mouse event is inside a given child widget.

Returns non-zero if the current Fl::event_x() and Fl::event_y() put it inside the given child widget's bounding box.

This method can only be used to check whether the mouse event is inside a **child** widget of the window that handles the event, and there must not be an intermediate subwindow (i.e. the widget must not be inside a subwindow of the current window). However, it is valid if the widget is inside a nested Fl_Group.

You must not use it with the window itself as the `o` argument in a window's handle() method.

Note

The mentioned restrictions are necessary, because this method does not transform coordinates of child widgets, and thus the given widget `o` must be within the *same* window that is handling the current event. Otherwise the results are undefined.

You should always call this rather than doing your own comparison so you are consistent about edge effects.

See Also

Fl::event_inside(int, int, int, int)

Parameters

`in`	*o*	child widget to be tested

Returns

non-zero, if mouse event is inside the widget

static int Fl::event_is_click ( ) `[inline]`, `[static]`

Returns non-zero if the mouse has not moved far enough and not enough time has passed since the last FL_PUSH or FL_KEYBOARD event for it to be considered a "drag" rather than a "click".

You can test this on FL_DRAG, FL_RELEASE, and FL_MOVE events.

static void Fl::event_is_click (int *i*) `[inline]`, `[static]`

Clears the value returned by Fl::event_is_click().

Useful to prevent the *next* click from being counted as a double-click or to make a popup menu pick an item with a single click. Don't pass non-zero to this.

static int Fl::event_key ( ) `[inline]`, `[static]`

Gets which key on the keyboard was last pushed.

The returned integer 'key code' is not necessarily a text equivalent for the keystroke. For instance: if someone presses '5' on the numeric keypad with numlock on, Fl::event_key() may return the 'key code' for this key, and NOT the character '5'. To always get the '5', use Fl::event_text() instead.

Returns

an integer 'key code', or 0 if the last event was not a key press or release.

See Also

int event_key(int), event_text(), compose(int&).

int Fl::event_key (int *key*) `[static]`

Returns true if the given `key` was held down (or pressed) *during* the last event.

This is constant until the next event is read from the server.

Fl::get_key(int) returns true if the given key is held down *now*. Under X this requires a round-trip to the server and is *much* slower than Fl::event_key(int).

Keys are identified by the *unshifted* values. FLTK defines a set of symbols that should work on most modern machines for every key on the keyboard:

- All keys on the main keyboard producing a printable ASCII character use the value of that ASCII character (as though shift, ctrl, and caps lock were not on). The space bar is 32.
- All keys on the numeric keypad producing a printable ASCII character use the value of that ASCII character plus FL_KP. The highest possible value is FL_KP_Last so you can range-check to see if something is on the keypad.
- All numbered function keys use the number on the function key plus FL_F. The highest possible number is FL_F_Last, so you can range-check a value.
- Buttons on the mouse are considered keys, and use the button number (where the left button is 1) plus FL_Button.
- All other keys on the keypad have a symbol: FL_Escape, FL_BackSpace, FL_Tab, FL_Enter, FL_-Print, FL_Scroll_Lock, FL_Pause, FL_Insert, FL_Home, FL_Page_Up, FL_Delete, FL_End, FL_Page-_Down, FL_Left, FL_Up, FL_Right, FL_Down, FL_Iso_Key, FL_Shift_L, FL_Shift_R, FL_Control_L, FL_Control_R, FL_Caps_Lock, FL_Alt_L, FL_Alt_R, FL_Meta_L, FL_Meta_R, FL_Menu, FL_Num_-Lock, FL_KP_Enter. Be careful not to confuse these with the very similar, but all-caps, symbols used by Fl::event_state().

On X Fl::get_key(FL_Button+n) does not work.

On WIN32 Fl::get_key(FL_KP_Enter) and Fl::event_key(FL_KP_Enter) do not work.

static int Fl::event_length ( ) `[inline]`, `[static]`

Returns the length of the text in Fl::event_text().

There will always be a nul at this position in the text. However there may be a nul before that if the keystroke translates to a nul character or you paste a nul character.

static int Fl::event_original_key ( ) `[inline]`, `[static]`

Returns the keycode of the last key event, regardless of the NumLock state.

If NumLock is deactivated, FLTK translates events from the numeric keypad into the corresponding arrow key events. event_key() returns the translated key code, whereas event_original_key() returns the keycode before NumLock translation.

static int Fl::event_shift ( ) `[inline]`, `[static]`

Returns non-zero if the Shift key is pressed.

static int Fl::event_state ( ) `[inline]`, `[static]`

Returns the keyboard and mouse button states of the last event.

This is a bitfield of what shift states were on and what mouse buttons were held down during the most recent event.

The legal event state bits are:

- FL_SHIFT
- FL_CAPS_LOCK
- FL_CTRL
- FL_ALT
- FL_NUM_LOCK
- FL_META
- FL_SCROLL_LOCK
- FL_BUTTON1
- FL_BUTTON2
- FL_BUTTON3

X servers do not agree on shift states, and FL_NUM_LOCK, FL_META, and FL_SCROLL_LOCK may not work. The values were selected to match the XFree86 server on Linux. In addition there is a bug in the way X works so that the shift state is not correctly reported until the first event *after* the shift key is pressed or released.

static int Fl::event_state (int *mask*) `[inline]`, `[static]`

Returns non-zero if any of the passed event state bits are turned on.

Use `mask` to pass the event states you're interested in. The legal event state bits are defined in Fl::event_state().

static const char∗ Fl::event_text ( ) `[inline], [static]`

Returns the text associated with the current event, including FL_PASTE or FL_DND_RELEASE events.

This can be used in response to FL_KEYUP, FL_KEYDOWN, FL_PASTE, and FL_DND_RELEASE.

When responding to FL_KEYUP/FL_KEYDOWN, use this function instead of Fl::event_key() to get the text equivalent of keystrokes suitable for inserting into strings and text widgets.

The returned string is guaranteed to be NULL terminated. However, see Fl::event_length() for the actual length of the string, in case the string itself contains NULLs that are part of the text data.

Returns

A NULL terminated text string equivalent of the last keystroke.

static int Fl::event_x_root ( ) `[inline], [static]`

Returns the mouse position on the screen of the event.

To find the absolute position of an Fl_Window on the screen, use the difference between event_x_-root(),event_y_root() and event_x(),event_y().

static int Fl::event_y_root ( ) `[inline], [static]`

Returns the mouse position on the screen of the event.

To find the absolute position of an Fl_Window on the screen, use the difference between event_x_-root(),event_y_root() and event_x(),event_y().

static Fl_Widget∗ Fl::focus ( ) `[inline], [static]`

Gets the current Fl::focus() widget.

See Also

Fl::focus(Fl_Widget∗)

void Fl::focus (Fl_Widget ∗ *o*) `[static]`

Sets the widget that will receive FL_KEYBOARD events.

If you change Fl::focus(), the previous widget and all parents (that don't contain the new widget) are sent FL_UNFOCUS events. Changing the focus does *not* send FL_FOCUS to this or any widget, because sending FL_FOCUS is supposed to *test* if the widget wants the focus (by it returning non-zero from handle()).

See Also

Fl_Widget::take_focus()

int Fl::get_key (int *key*) `[static]`

Returns true if the given `key` is held down *now*.

Under X this requires a round-trip to the server and is *much* slower than Fl::event_key(int).

See Also

event_key(int)

static void Fl::get_mouse (int & , int &) `[static]`

Return where the mouse is on the screen by doing a round-trip query to the server.

You should use Fl::event_x_root() and Fl::event_y_root() if possible, but this is necessary if you are not sure if a mouse event has been processed recently (such as to position your first window). If the display is not open, this will open it.

int Fl::handle (int *e*, Fl_Window ∗ *window*) `[static]`

Handle events from the window system.

This is called from the native event dispatch after native events have been converted to FLTK notation. This function calls Fl::handle_(int, Fl_Window∗) unless the user sets a dispatch function. If a user dispatch function is set, the user must make sure that Fl::handle_() is called, or the event will be ignored.

Parameters

e	the event type (Fl::event_number() is not yet set)
window	the window that caused this event

Returns

0 if the event was not handled

See Also

Fl::add_handler(Fl_Event_Handler)
Fl::event_dispatch(Fl_Event_Dispatch)

int Fl::handle_ (int *e*, Fl_Window ∗ *window*) `[static]`

Handle events from the window system.

This function is called from the native event dispatch, unless the user sets another dispatch function. In that case, the user dispatch function must decide when to call Fl::handle_(int, Fl_Window∗)

Parameters

e	the event type (Fl::event_number() is not yet set)
window	the window that caused this event

Returns

0 if the event was not handled

See Also

Fl::event_dispatch(Fl_Event_Dispatch)

static Fl_Widget∗ Fl::pushed ( ) `[inline]`, `[static]`

Gets the widget that is being pushed.

See Also

void pushed(Fl_Widget∗)

void Fl::pushed (Fl_Widget ∗ *o*) `[static]`

Sets the widget that is being pushed.

FL_DRAG or FL_RELEASE (and any more FL_PUSH) events will be sent to this widget.

If you change the pushed widget, the previous one and all parents (that don't contain the new widget) are sent FL_RELEASE events. Changing this does *not* send FL_PUSH to this or any widget, because sending FL_PUSH is supposed to *test* if the widget wants the mouse (by it returning non-zero from handle()).

void Fl::remove_handler (Fl_Event_Handler *ha*) `[static]`

Removes a previously added event handler.

See Also

Fl::handle(int, Fl_Window∗)

void Fl::remove_system_handler (Fl_System_Handler *ha*) `[static]`

Removes a previously added system event handler.

Parameters

ha	The event handler function to remove

See Also

Fl::add_system_handler(Fl_System_Handler)

int Fl::test_shortcut (Fl_Shortcut *shortcut*) `[static]`

Tests the current event, which must be an FL_KEYBOARD or FL_SHORTCUT, against a shortcut value (described in Fl_Button).

Not to be confused with Fl_Widget::test_shortcut().

Returns

non-zero if there is a match.

30.3.3 Variable Documentation

const char∗ const fl_eventnames[ ]

This is an array of event names you can use to convert event numbers into names.

The array gets defined inline wherever your '#include <FL/names.h>' appears.

Example:

```
#include <FL/names.h>           // array will be defined here
int MyClass::handle(int e) {
    printf("Event was %s (%d)\n", fl_eventnames[e], e);
    // ..resulting output might be e.g. "Event was FL_PUSH (1)"..
    [..]
}
```

const char∗ const fl_fontnames[ ]

Initial value:

```
=
{
  "FL_HELVETICA",
  "FL_HELVETICA_BOLD",
  "FL_HELVETICA_ITALIC",
  "FL_HELVETICA_BOLD_ITALIC",
  "FL_COURIER",
  "FL_COURIER_BOLD",
  "FL_COURIER_ITALIC",
  "FL_COURIER_BOLD_ITALIC",
  "FL_TIMES",
  "FL_TIMES_BOLD",
  "FL_TIMES_ITALIC",
  "FL_TIMES_BOLD_ITALIC",
  "FL_SYMBOL",
  "FL_SCREEN",
  "FL_SCREEN_BOLD",
  "FL_ZAPF_DINGBATS",
}
```

This is an array of font names you can use to convert font numbers into names.
The array gets defined inline wherever your '#include <FL/names.h>' appears.
Example:

```
#include <FL/names.h>           // array will be defined here
int MyClass::my_callback(Fl_Widget *w, void*) {
    int fnum = w->labelfont();
    // Resulting output might be e.g. "Label's font is FL_HELVETICA (0)"
    printf("Label's font is %s (%d)\n", fl_fontnames[fnum], fnum);
    // ..resulting output might be e.g. "Label's font is FL_HELVETICA (0)"..
    [..]
}
```

30.4 Selection & Clipboard functions

FLTK global copy/cut/paste functions declared in <FL/Fl.H>

Functions

- static void Fl::add_clipboard_notify (Fl_Clipboard_Notify_Handler h, void ∗data=0)

  FLTK will call the registered callback whenever there is a change to the selection buffer or the clipboard.
- static int Fl::clipboard_contains (const char ∗type)

  Returns non 0 if the clipboard contains data matching `type`.
- static void Fl::copy (const char ∗stuff, int len, int destination=0, const char ∗type=Fl::clipboard_-plain_text)

  Copies the data pointed to by `stuff` to the selection buffer (`destination` is 0), the clipboard (`destination` is 1), or both (`destination` is 2).
- static int Fl::dnd ()

  Initiate a Drag And Drop operation.
- static void Fl::paste (Fl_Widget &receiver, int source, const char ∗type=Fl::clipboard_plain_text)

  Pastes the data from the selection buffer (`source` is 0) or the clipboard (`source` is 1) into `receiver`.
- static void Fl::paste (Fl_Widget &receiver)

  Backward compatibility only.
- static void Fl::remove_clipboard_notify (Fl_Clipboard_Notify_Handler h)

  Stop calling the specified callback when there are changes to the selection buffer or the clipboard.
- static void Fl::selection (Fl_Widget &owner, const char ∗, int len)

  Changes the current selection.
- static Fl_Widget ∗ Fl::selection_owner ()

  back-compatibility only: Gets the widget owning the current selection
- static void Fl::selection_owner (Fl_Widget ∗)

  Back-compatibility only: The single-argument call can be used to move the selection to another widget or to set the owner to NULL, without changing the actual text of the selection.

Variables

- static char const ∗const Fl::clipboard_image = "image"

  Denotes image data.
- static char const ∗const Fl::clipboard_plain_text = "text/plain"

  Denotes plain textual data.

30.4.1 Detailed Description

FLTK global copy/cut/paste functions declared in <FL/Fl.H>

30.4.2 Function Documentation

void Fl::add_clipboard_notify (Fl_Clipboard_Notify_Handler *h*, void ∗ *data* = `0`) `[static]`

FLTK will call the registered callback whenever there is a change to the selection buffer or the clipboard.

The source argument indicates which of the two has changed. Only changes by other applications are reported.

Example:

```
void clip_callback(int source, void *data) {
    if ( source == 0 ) printf("CLIP CALLBACK: selection buffer changed\n");
    if ( source == 1 ) printf("CLIP CALLBACK: clipboard changed\n");
}
[..]
int main() {
    [..]
    Fl::add_clipboard_notify(clip_callback);
    [..]
}
```

Note

> Some systems require polling to monitor the clipboard and may therefore have some delay in detecting changes.

static int Fl::clipboard_contains (const char * *type*) `[static]`

Returns non 0 if the clipboard contains data matching `type`.

`type` can be Fl::clipboard_plain_text or Fl::clipboard_image.

static void Fl::copy (const char * *stuff,* int *len,* int *destination* = *0,* const char * *type* = Fl::clipboard_plain_text) `[static]`

Copies the data pointed to by `stuff` to the selection buffer (`destination` is 0), the clipboard (`destination` is 1), or both (`destination` is 2).

Copying to both is only relevant on X11, on other platforms it maps to the clipboard (1). `len` is the number of relevant bytes in `stuff`. `type` is always Fl::clipboard_plain_text. The selection buffer is used for middle-mouse pastes and for drag-and-drop selections. The clipboard is used for traditional copy/cut/paste operations.

Note

> This function is, at present, intended only to copy UTF-8 encoded textual data. To copy graphical data, use the Fl_Copy_Surface class. The `type` argument may allow in the future to copy other kinds of data.

int Fl::dnd ( ) `[static]`

Initiate a Drag And Drop operation.

The selection buffer should be filled with relevant data before calling this method. FLTK will then initiate the system wide drag and drop handling. Dropped data will be marked as *text*.

Create a selection first using: Fl::copy(const char *stuff, int len, 0)

static void Fl::paste (Fl_Widget & *receiver,* int *source,* const char * *type* = Fl::clipboard_plain_text) `[static]`

Pastes the data from the selection buffer (`source` is 0) or the clipboard (`source` is 1) into `receiver`.

The selection buffer (`source` is 0) is used for middle-mouse pastes and for drag-and-drop selections. The clipboard (`source` is 1) is used for copy/cut/paste operations.

If `source` is 1, the optional `type` argument indicates what type of data is requested from the clipboard. At present, Fl::clipboard_plain_text (requesting text data) and Fl::clipboard_image (requesting image data) are possible. Set things up so the handle function of the `receiver` widget will be called with an FL_PASTE event some time in the future if the clipboard does contain data of the requested type. While processing the FL_PASTE event:

- if `type` is Fl::clipboard_plain_text, the text string from the specified `source` is in Fl::event_text() with UTF-8 encoding, and the number of bytes in Fl::event_length(). If Fl::paste() gets called during the drop step of a files-drag-and-drop operation, Fl::event_text() contains a list of filenames (see Drag and Drop Events).
- if `type` is Fl::clipboard_image, the pointer returned by Fl::event_clipboard() can be safely cast to type Fl_Image ∗ to obtain a pointer to the pasted image. Furthermore, starting with FLTK 1.3.4, the image is of type Fl_RGB_Image across all platforms. If `receiver` accepts the clipboard image, receiver.handle() should return 1 and the application should take ownership of this image (that is, delete it after use). Conversely, if receiver.handle() returns 0, the application must not use the image.

The receiver should be prepared to be called *directly* by this, or for it to happen *later*, or possibly *not at all*. This allows the window system to take as long as necessary to retrieve the paste buffer (or even to screw up completely) without complex and error-prone synchronization code in FLTK.

Platform details for image data:

- Unix/Linux platform: Clipboard images in PNG or BMP formats are recognized. Requires linking with the fltk_images library.
- MSWindows platform: Both bitmap and vectorial (Enhanced metafile) data from clipboard can be pasted as image data.
- Mac OS X platform: Both bitmap (TIFF) and vectorial (PDF) data from clipboard can be pasted as image data.

void Fl::paste (Fl_Widget & *receiver*) `[static]`

Backward compatibility only.

This calls Fl::paste(receiver, 0);

See Also

Fl::paste(Fl_Widget &receiver, int clipboard, const char∗ type)

void Fl::selection (Fl_Widget & *owner*, const char ∗ *text*, int *len*) `[static]`

Changes the current selection.

The block of text is copied to an internal buffer by FLTK (be careful if doing this in response to an FL_PASTE as this *may* be the same buffer returned by event_text()). The selection_owner() widget is set to the passed owner.

static Fl_Widget∗ Fl::selection_owner ( ) `[inline]`, `[static]`

back-compatibility only: Gets the widget owning the current selection

See Also

Fl_Widget∗ selection_owner(Fl_Widget∗)

void Fl::selection_owner (Fl_Widget ∗ *owner*) `[static]`

Back-compatibility only: The single-argument call can be used to move the selection to another widget or to set the owner to NULL, without changing the actual text of the selection.

FL_SELECTIONCLEAR is sent to the previous selection owner, if any.

Copying the buffer every time the selection is changed is obviously wasteful, especially for large selections. An interface will probably be added in a future version to allow the selection to be made by a callback function. The current interface will be emulated on top of this.

30.5 Screen functions

fl global screen functions declared in <FL/Fl.H>

Functions

- static int Fl::h ()

  Returns the height in pixels of the main screen work area.
- static int Fl::screen_count ()

  Gets the number of available screens.
- static void Fl::screen_dpi (float &h, float &v, int n=0)

  Gets the screen resolution in dots-per-inch for the given screen.
- static int Fl::screen_num (int x, int y)

  Gets the screen number of a screen that contains the specified screen position `x, y`.
- static int Fl::screen_num (int x, int y, int w, int h)

  Gets the screen number for the screen which intersects the most with the rectangle defined by `x, y, w, h`.
- static void Fl::screen_work_area (int &X, int &Y, int &W, int &H, int mx, int my)

  Gets the bounding box of the work area of a screen that contains the specified screen position `mx, my`.
- static void Fl::screen_work_area (int &X, int &Y, int &W, int &H, int n)

  Gets the bounding box of the work area of the given screen.
- static void Fl::screen_work_area (int &X, int &Y, int &W, int &H)

  Gets the bounding box of the work area of the screen that contains the mouse pointer.
- static void Fl::screen_xywh (int &X, int &Y, int &W, int &H)

  Gets the bounding box of a screen that contains the mouse pointer.
- static void Fl::screen_xywh (int &X, int &Y, int &W, int &H, int mx, int my)

  Gets the bounding box of a screen that contains the specified screen position `mx, my`.
- static void Fl::screen_xywh (int &X, int &Y, int &W, int &H, int n)

  Gets the screen bounding rect for the given screen.
- static void Fl::screen_xywh (int &X, int &Y, int &W, int &H, int mx, int my, int mw, int mh)

  Gets the screen bounding rect for the screen which intersects the most with the rectangle defined by `mx, my, mw, mh`.
- static int Fl::w ()

  Returns the width in pixels of the main screen work area.
- static int Fl::x ()

  Returns the leftmost x coordinate of the main screen work area.
- static int Fl::y ()

  Returns the topmost y coordinate of the main screen work area.

30.5.1 Detailed Description

fl global screen functions declared in <FL/Fl.H>

30.5.2 Function Documentation

static int Fl::h ( ) `[static]`

Returns the height in pixels of the main screen work area.

void Fl::screen_dpi (float & *h*, float & *v*, int *n* = `0`) `[static]`

Gets the screen resolution in dots-per-inch for the given screen.

Parameters

out	*h,v*	horizontal and vertical resolution
in	*n*	the screen number (0 to Fl::screen_count() - 1)

See Also

void screen_xywh(int &x, int &y, int &w, int &h, int mx, int my)

int Fl::screen_num (int *x*, int *y*) `[static]`

Gets the screen number of a screen that contains the specified screen position `x`, `y`.
Parameters

in	*x,y*	the absolute screen position

int Fl::screen_num (int *x*, int *y*, int *w*, int *h*) `[static]`

Gets the screen number for the screen which intersects the most with the rectangle defined by `x`, `y`, `w`, `h`.
Parameters

in	*x,y,w,h*	the rectangle to search for intersection with

void Fl::screen_work_area (int & *X*, int & *Y*, int & *W*, int & *H*, int *mx*, int *my*) `[static]`

Gets the bounding box of the work area of a screen that contains the specified screen position `mx`, `my`.
Parameters

out	*X,Y,W,H*	the work area bounding box
in	*mx,my*	the absolute screen position

void Fl::screen_work_area (int & *X*, int & *Y*, int & *W*, int & *H*, int *n*) `[static]`

Gets the bounding box of the work area of the given screen.
Parameters

out	*X,Y,W,H*	the work area bounding box
in	*n*	the screen number (0 to Fl::screen_count() - 1)

See Also

void screen_xywh(int &x, int &y, int &w, int &h, int mx, int my)

static void Fl::screen_work_area (int & *X*, int & *Y*, int & *W*, int & *H*) `[inline]`, `[static]`

Gets the bounding box of the work area of the screen that contains the mouse pointer.
Parameters

out	*X,Y,W,H*	the work area bounding box

See Also

void screen_work_area(int &x, int &y, int &w, int &h, int mx, int my)

static void Fl::screen_xywh (int & *X*, int & *Y*, int & *W*, int & *H*) `[inline]`, `[static]`

Gets the bounding box of a screen that contains the mouse pointer.

Parameters

out	*X,Y,W,H*	the corresponding screen bounding box

See Also

void screen_xywh(int &x, int &y, int &w, int &h, int mx, int my)

void Fl::screen_xywh (int & *X*, int & *Y*, int & *W*, int & *H*, int *mx*, int *my*) `[static]`

Gets the bounding box of a screen that contains the specified screen position `mx`, `my`.
Parameters

out	*X,Y,W,H*	the corresponding screen bounding box
in	*mx,my*	the absolute screen position

void Fl::screen_xywh (int & *X*, int & *Y*, int & *W*, int & *H*, int *n*) `[static]`

Gets the screen bounding rect for the given screen.
Under MSWindows, Mac OS X, and the Gnome desktop, screen #0 contains the menubar/taskbar
Parameters

out	*X,Y,W,H*	the corresponding screen bounding box
in	*n*	the screen number (0 to Fl::screen_count() - 1)

See Also

void screen_xywh(int &x, int &y, int &w, int &h, int mx, int my)

void Fl::screen_xywh (int & *X*, int & *Y*, int & *W*, int & *H*, int *mx*, int *my*, int *mw*, int *mh*) `[static]`

Gets the screen bounding rect for the screen which intersects the most with the rectangle defined by `mx`, `my`, `mw`, `mh`.
Parameters

out	*X,Y,W,H*	the corresponding screen bounding box
in	*mx,my,mw,mh*	the rectangle to search for intersection with

See Also

void screen_xywh(int &X, int &Y, int &W, int &H, int n)

static int Fl::w ( ) `[static]`

Returns the width in pixels of the main screen work area.

static int Fl::x ( ) `[static]`

Returns the leftmost x coordinate of the main screen work area.

static int Fl::y ( ) `[static]`

Returns the topmost y coordinate of the main screen work area.

30.6 Color & Font functions

fl global color, font functions.

Functions

- void fl_color (Fl_Color c)

  Sets the color for all subsequent drawing operations.
- void fl_color (int c)

  for back compatibility - use fl_color(Fl_Color c) instead
- void fl_color (uchar r, uchar g, uchar b)

  Sets the color for all subsequent drawing operations.
- Fl_Color fl_color ()

  Returns the last fl_color() that was set.
- Fl_Color fl_color_average (Fl_Color color1, Fl_Color color2, float weight)

  Returns the weighted average color between the two given colors.
- Fl_Color fl_contrast (Fl_Color fg, Fl_Color bg)

  Returns a color that contrasts with the background color.
- int fl_descent ()

  Returns the recommended distance above the bottom of a fl_height() tall box to draw the text at so it looks centered vertically in that box.
- void fl_font (Fl_Font face, Fl_Fontsize fsize)

  Sets the current font, which is then used in various drawing routines.
- Fl_Font fl_font ()

  Returns the `face` set by the most recent call to fl_font().
- int fl_height ()

  Returns the recommended minimum line spacing for the current font.
- FL_EXPORT int fl_height (int font, int size)

  This function returns the actual height of the specified `font` and `size`.
- Fl_Color fl_inactive (Fl_Color c)

  Returns the inactive, dimmed version of the given color.
- FL_EXPORT const char ∗ fl_latin1_to_local (const char ∗t, int n=-1)

  Converts text from Windows/X11 latin1 character set to local encoding.
- FL_EXPORT const char ∗ fl_local_to_latin1 (const char ∗t, int n=-1)

  Converts text from local encoding to Windowx/X11 latin1 character set.
- FL_EXPORT const char ∗ fl_local_to_mac_roman (const char ∗t, int n=-1)

  Converts text from local encoding to Mac Roman character set.
- FL_EXPORT const char ∗ fl_mac_roman_to_local (const char ∗t, int n=-1)

  Converts text from Mac Roman character set to local encoding.
- FL_EXPORT Fl_Color fl_show_colormap (Fl_Color oldcol)

  Pops up a window to let the user pick a colormap entry.
- Fl_Fontsize fl_size ()

  Returns the `size` set by the most recent call to fl_font().
- FL_EXPORT void fl_text_extents (const char ∗, int &dx, int &dy, int &w, int &h)

  Determines the minimum pixel dimensions of a nul-terminated string.
- void fl_text_extents (const char ∗t, int n, int &dx, int &dy, int &w, int &h)

  Determines the minimum pixel dimensions of a sequence of `n` characters.
- FL_EXPORT double fl_width (const char ∗txt)

Returns the typographical width of a nul-terminated string using the current font face and size.

- double fl_width (const char ∗txt, int n)

  Returns the typographical width of a sequence of `n` *characters using the current font face and size.*
- double fl_width (unsigned int c)

  Returns the typographical width of a single character using the current font face and size.
- ulong fl_xpixel (uchar r, uchar g, uchar b)

  Returns the X pixel number used to draw the given rgb color.
- ulong fl_xpixel (Fl_Color i)

  Returns the X pixel number used to draw the given FLTK color index.
- static void Fl::free_color (Fl_Color i, int overlay=0)

  Frees the specified color from the colormap, if applicable.
- static unsigned Fl::get_color (Fl_Color i)

  Returns the RGB value(s) for the given FLTK color index.
- static void Fl::get_color (Fl_Color i, uchar &red, uchar &green, uchar &blue)

  Returns the RGB value(s) for the given FLTK color index.
- static const char ∗ Fl::get_font (Fl_Font)

  Gets the string for this face.
- static const char ∗ Fl::get_font_name (Fl_Font, int ∗attributes=0)

  Get a human-readable string describing the family of this face.
- static int Fl::get_font_sizes (Fl_Font, int ∗&sizep)

  Return an array of sizes in `sizep`*.*
- static void Fl::set_color (Fl_Color, uchar, uchar, uchar)

  Sets an entry in the fl_color index table.
- static void Fl::set_color (Fl_Color i, unsigned c)

  Sets an entry in the fl_color index table.
- static void Fl::set_font (Fl_Font, const char ∗)

  Changes a face.
- static void Fl::set_font (Fl_Font, Fl_Font)

  Copies one face to another.
- static Fl_Font Fl::set_fonts (const char ∗=0)

  FLTK will open the display, and add every fonts on the server to the face table.

30.6.1 Detailed Description

fl global color, font functions. These functions are declared in <FL/Fl.H> or <FL/fl_draw.H>.

30.6.2 Function Documentation

void fl_color (Fl_Color *c*) `[inline]`

Sets the color for all subsequent drawing operations.

For colormapped displays, a color cell will be allocated out of `fl_colormap` the first time you use a color. If the colormap fills up then a least-squares algorithm is used to find the closest color. If no valid graphical context (fl_gc) is available, the foreground is not set for the current window.

Parameters

in	*c*	color

void fl_color (uchar *r*, uchar *g*, uchar *b*) `[inline]`

Sets the color for all subsequent drawing operations.

The closest possible match to the RGB color is used. The RGB color is used directly on TrueColor displays. For colormap visuals the nearest index in the gray ramp or color cube is used. If no valid graphical context (fl_gc) is available, the foreground is not set for the current window.

Parameters

in	*r,g,b*	color components

Fl_Color fl_color (void) `[inline]`

Returns the last fl_color() that was set.

This can be used for state save/restore.

Fl_Color fl_color_average (Fl_Color *color1*, Fl_Color *color2*, float *weight*)

Returns the weighted average color between the two given colors.

The red, green and blue values are averages using the following formula:

```
color = color1 * weight  + color2 * (1 - weight)
```

Thus, a `weight` value of 1.0 will return the first color, while a value of 0.0 will return the second color.

Parameters

in	*color1,color2*	boundary colors
in	*weight*	weighting factor

Fl_Color fl_contrast (Fl_Color *fg*, Fl_Color *bg*)

Returns a color that contrasts with the background color.

This will be the foreground color if it contrasts sufficiently with the background color. Otherwise, returns `FL_WHITE` or `FL_BLACK` depending on which color provides the best contrast.

Parameters

in	*fg,bg*	foreground and background colors

Returns

contrasting color

void fl_font (Fl_Font *face*, Fl_Fontsize *fsize*) `[inline]`

Sets the current font, which is then used in various drawing routines.

You may call this outside a draw context if necessary to call fl_width(), but on X this will open the display.

The font is identified by a `face` and a `size`. The size of the font is measured in pixels and not ”points”. Lines should be spaced `size` pixels apart or more.

Fl_Font fl_font (void) `[inline]`

Returns the `face` set by the most recent call to fl_font().

This can be used to save/restore the font.

int fl_height ( ) `[inline]`

Returns the recommended minimum line spacing for the current font.

You can also use the value of `size` passed to fl_font()

FL_EXPORT int fl_height (int *font*, int *size*)

This function returns the actual height of the specified `font` and `size`.

Normally the font height should always be 'size', but with the advent of XFT, there are (currently) complexities that seem to only be solved by asking the font what its actual font height is. (See STR#2115)

This function was originally undocumented in 1.1.x, and was used only by Fl_Text_Display. We're now documenting it in 1.3.x so that apps that need precise height info can get it with this function.

Returns

the height of the font in pixels.

Todo In the future, when the XFT issues are resolved, this function should simply return the 'size' value.

FL_EXPORT const char∗ fl_latin1_to_local (const char ∗ *t*, int *n* = `-1`)

Converts text from Windows/X11 latin1 character set to local encoding.

Parameters

`in`	*t*	character string (latin1 encoding)
`in`	*n*	optional number of characters to convert (default is all)

Returns

pointer to internal buffer containing converted characters

FL_EXPORT const char∗ fl_local_to_latin1 (const char ∗ *t*, int *n* = `-1`)

Converts text from local encoding to Windowx/X11 latin1 character set.

Parameters

`in`	*t*	character string (local encoding)
`in`	*n*	optional number of characters to convert (default is all)

Returns

pointer to internal buffer containing converted characters

FL_EXPORT const char∗ fl_local_to_mac_roman (const char ∗ *t*, int *n* = `-1`)

Converts text from local encoding to Mac Roman character set.

Parameters

`in`	*t*	character string (local encoding)
`in`	*n*	optional number of characters to convert (default is all)

Returns

pointer to internal buffer containing converted characters

FL_EXPORT const char∗ fl_mac_roman_to_local (const char ∗ *t*, int *n* = `-1`)

Converts text from Mac Roman character set to local encoding.
Parameters

`in`	*t*	character string (Mac Roman encoding)
`in`	*n*	optional number of characters to convert (default is all)

Returns

pointer to internal buffer containing converted characters

FL_EXPORT Fl_Color fl_show_colormap (Fl_Color *oldcol*)

Pops up a window to let the user pick a colormap entry.

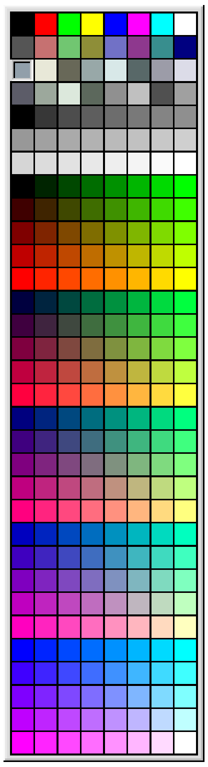


Figure 30.1: fl_show_colormap

Parameters

`in`	*oldcol*	color to be highlighted when grid is shown.

Return values

Fl_Color	value of the chosen colormap entry.

See Also

Fl_Color_Chooser

Fl_Fontsize fl_size ( ) `[inline]`

Returns the `size` set by the most recent call to fl_font().
This can be used to save/restore the font.

FL_EXPORT void fl_text_extents (const char ∗ , int & *dx*, int & *dy*, int & *w*, int & *h*)

Determines the minimum pixel dimensions of a nul-terminated string.
Usage: given a string "txt" drawn using fl_draw(txt, x, y) you would determine its pixel extents on the display using fl_text_extents(txt, dx, dy, wo, ho) such that a bounding box that exactly fits around the text could be drawn with fl_rect(x+dx, y+dy, wo, ho). Note the dx, dy values hold the offset of the first "colored in" pixel of the string, from the draw origin.
No FLTK symbol expansion will be performed.

void fl_text_extents (const char ∗ *t*, int *n*, int & *dx*, int & *dy*, int & *w*, int & *h*) `[inline]`

Determines the minimum pixel dimensions of a sequence of `n` characters.

See Also

fl_text_extents(const char∗, int& dx, int& dy, int& w, int& h)

FL_EXPORT double fl_width (const char ∗ *txt*)

Returns the typographical width of a nul-terminated string using the current font face and size.

double fl_width (const char ∗ *txt*, int *n*) `[inline]`

Returns the typographical width of a sequence of `n` characters using the current font face and size.

double fl_width (unsigned int *c*) `[inline]`

Returns the typographical width of a single character using the current font face and size.

Note

if a valid fl_gc is NOT found then it uses the first window gc, or the screen gc if no fltk window is available when called.

ulong fl_xpixel (uchar *r*, uchar *g*, uchar *b*)

Returns the X pixel number used to draw the given rgb color.
This is the X pixel that fl_color() would use.

Parameters

`in`	*r,g,b*	color components

Returns

X pixel number

ulong fl_xpixel (Fl_Color *i*)

Returns the X pixel number used to draw the given FLTK color index.

This is the X pixel that fl_color() would use.

Parameters

`in`	*i*	color index

Returns

X pixel number

void Fl::free_color (Fl_Color *i*, int *overlay* = `0`) `[static]`

Frees the specified color from the colormap, if applicable.

Free color `i` if used, and clear mapping table entry.

If overlay is non-zero then the color is freed from the overlay colormap.

Parameters

`in`	*i*	color index
`in`	*overlay*	0 for normal, 1 for overlay color

unsigned Fl::get_color (Fl_Color *i*) `[static]`

Returns the RGB value(s) for the given FLTK color index.

This form returns the RGB values packed in a 32-bit unsigned integer with the red value in the upper 8 bits, the green value in the next 8 bits, and the blue value in bits 8-15. The lower 8 bits will always be 0.

void Fl::get_color (Fl_Color *i*, uchar & *red*, uchar & *green*, uchar & *blue*) `[static]`

Returns the RGB value(s) for the given FLTK color index.

This form returns the red, green, and blue values separately in referenced variables.

See also unsigned get_color(Fl_Color c)

const char ∗ Fl::get_font (Fl_Font *fnum*) `[static]`

Gets the string for this face.

This string is different for each face. Under X this value is passed to XListFonts to get all the sizes of this face.

const char ∗ Fl::get_font_name (Fl_Font *fnum*, int ∗ *attributes* = `0`) `[static]`

Get a human-readable string describing the family of this face.

This is useful if you are presenting a choice to the user. There is no guarantee that each face has a different name. The return value points to a static buffer that is overwritten each call.

The integer pointed to by `attributes` (if the pointer is not zero) is set to zero, FL_BOLD or FL_ITALIC or FL_BOLD | FL_ITALIC. To locate a "family" of fonts, search forward and back for a set with non-zero attributes, these faces along with the face with a zero attribute before them constitute a family.

int Fl::get_font_sizes (Fl_Font *fnum*, int ∗& *sizep*) `[static]`

Return an array of sizes in `sizep`.

The return value is the length of this array. The sizes are sorted from smallest to largest and indicate what sizes can be given to fl_font() that will be matched exactly (fl_font() will pick the closest size for other sizes). A zero in the first location of the array indicates a scalable font, where any size works, although the array may list sizes that work "better" than others. Warning: the returned array points at a static buffer that is overwritten each call. Under X this will open the display.

void Fl::set_color (Fl_Color *i*, uchar *red*, uchar *green*, uchar *blue*) `[static]`

Sets an entry in the fl_color index table.

You can set it to any 8-bit RGB color. The color is not allocated until fl_color(i) is used.

void Fl::set_color (Fl_Color *i*, unsigned *c*) `[static]`

Sets an entry in the fl_color index table.

Set color mapping table entry `i` to color `c`.

You can set it to any 8-bit RGB color. The color is not allocated until fl_color(i) is used.

Parameters

`in`	*i*	color index
`in`	*c*	color

void Fl::set_font (Fl_Font *fnum*, const char ∗ *name*) `[static]`

Changes a face.

The string pointer is simply stored, the string is not copied, so the string must be in static memory.

void Fl::set_font (Fl_Font *fnum*, Fl_Font *from*) `[static]`

Copies one face to another.

Fl_Font Fl::set_fonts (const char ∗ *xstarname* = *0*) `[static]`

FLTK will open the display, and add every fonts on the server to the face table.

It will attempt to put "families" of faces together, so that the normal one is first, followed by bold, italic, and bold italic.

The optional argument is a string to describe the set of fonts to add. Passing NULL will select only fonts that have the ISO8859-1 character set (and are thus usable by normal text). Passing "-∗" will select all fonts with any encoding as long as they have normal X font names with dashes in them. Passing "∗" will list every font that exists (on X this may produce some strange output). Other values may be useful but are system dependent. With WIN32 NULL selects fonts with ISO8859-1 encoding and non-NULL selects all fonts.

The return value is how many faces are in the table after this is done.

30.7 Drawing functions

FLTK global graphics and GUI drawing functions.

Macros

- #define fl_clip fl_push_clip

    Intersects the current clip region with a rectangle and pushes this new region onto the stack (deprecated).

Enumerations

- enum {
  FL_SOLID = 0, FL_DASH = 1, FL_DOT = 2, FL_DASHDOT = 3,
  FL_DASHDOTDOT = 4, FL_CAP_FLAT = 0x100, FL_CAP_ROUND = 0x200, FL_CAP_SQUARE = 0x300,
  FL_JOIN_MITER = 0x1000, FL_JOIN_ROUND = 0x2000, FL_JOIN_BEVEL = 0x3000 }

Functions

- void Fl_Quartz_Graphics_Driver::copy_offscreen (int x, int y, int w, int h, Fl_Offscreen pixmap, int srcx, int srcy)

    see fl_copy_offscreen()
- FL_EXPORT int fl_add_symbol (const char ∗name, void(∗drawit)(Fl_Color), int scalable)

    Adds a symbol to the system.
- void fl_arc (int x, int y, int w, int h, double a1, double a2)

    Draw ellipse sections using integer coordinates.
- void fl_arc (double x, double y, double r, double start, double end)

    Adds a series of points to the current path on the arc of a circle.
- void fl_begin_complex_polygon ()

    Starts drawing a complex filled polygon.
- void fl_begin_line ()

    Starts drawing a list of lines.
- void fl_begin_loop ()

    Starts drawing a closed sequence of lines.
- void fl_begin_offscreen (Fl_Offscreen ctx)

    Send all subsequent drawing commands to this offscreen buffer.
- void fl_begin_points ()

    Starts drawing a list of points.
- void fl_begin_polygon ()

    Starts drawing a convex filled polygon.
- FL_EXPORT char fl_can_do_alpha_blending ()

    Checks whether platform supports true alpha blending for RGBA images.
- FL_EXPORT void fl_chord (int x, int y, int w, int h, double a1, double a2)

    fl_chord declaration is a place holder - the function does not yet exist
- void fl_circle (double x, double y, double r)

    fl_circle() is equivalent to fl_arc(x,y,r,0,360), but may be faster.
- int fl_clip_box (int x, int y, int w, int h, int &X, int &Y, int &W, int &H)

    Intersects the rectangle with the current clip region and returns the bounding box of the result.
- void fl_clip_region (Fl_Region r)

Replaces the top of the clipping stack with a clipping region of any shape.

- Fl_Region fl_clip_region ()

  Returns the current clipping region.
- void fl_copy_offscreen (int x, int y, int w, int h, Fl_Offscreen pixmap, int srcx, int srcy)

  Copy a rectangular area of the given offscreen buffer into the current drawing destination.
- Fl_Offscreen fl_create_offscreen (int w, int h)

  Creation of an offscreen graphics buffer.
- FL_EXPORT void fl_cursor (Fl_Cursor)

  Sets the cursor for the current window to the specified shape and colors.
- FL_EXPORT void **fl_cursor** (Fl_Cursor, Fl_Color fg, Fl_Color bg=FL_WHITE)
- void fl_curve (double X0, double Y0, double X1, double Y1, double X2, double Y2, double X3, double Y3)

  Adds a series of points on a Bezier curve to the path.
- void fl_delete_offscreen (Fl_Offscreen ctx)

  Deletion of an offscreen graphics buffer.
- FL_EXPORT void fl_draw (const char ∗str, int x, int y)

  Draws a nul-terminated UTF-8 string starting at the given `x, y` location.
- FL_EXPORT void fl_draw (int angle, const char ∗str, int x, int y)

  Draws a nul-terminated UTF-8 string starting at the given `x, y` location and rotating `angle` degrees counter-clockwise.
- void fl_draw (const char ∗str, int n, int x, int y)

  Draws starting at the given `x, y` location a UTF-8 string of length `n` bytes.
- void fl_draw (int angle, const char ∗str, int n, int x, int y)

  Draws at the given `x, y` location a UTF-8 string of length `n` bytes rotating `angle` degrees counter-clockwise.
- FL_EXPORT void fl_draw (const char ∗str, int x, int y, int w, int h, Fl_Align align, Fl_Image ∗img=0, int draw_symbols=1)

  Fancy string drawing function which is used to draw all the labels.
- FL_EXPORT void fl_draw (const char ∗str, int x, int y, int w, int h, Fl_Align align, void(∗callthis)(const char ∗, int, int, int), Fl_Image ∗img=0, int draw_symbols=1)

  The same as fl_draw(const char∗,int,int,int,int,Fl_Align,Fl_Image∗,int) with the addition of the `callthis` parameter, which is a pointer to a text drawing function such as fl_draw(const char∗, int, int, int) to do the real work.
- FL_EXPORT void fl_draw_box (Fl_Boxtype, int x, int y, int w, int h, Fl_Color)

  Draws a box using given type, position, size and color.
- void fl_draw_image (const uchar ∗buf, int X, int Y, int W, int H, int D=3, int L=0)

  Draws an 8-bit per color RGB or luminance image.
- void fl_draw_image (Fl_Draw_Image_Cb cb, void ∗data, int X, int Y, int W, int H, int D=3)

  Draws an image using a callback function to generate image data.
- void fl_draw_image_mono (const uchar ∗buf, int X, int Y, int W, int H, int D=1, int L=0)

  Draws a gray-scale (1 channel) image.
- void fl_draw_image_mono (Fl_Draw_Image_Cb cb, void ∗data, int X, int Y, int W, int H, int D=1)

  Draws a gray-scale image using a callback function to generate image data.
- FL_EXPORT int fl_draw_pixmap (char ∗const ∗data, int x, int y, Fl_Color=FL_GRAY)

  Draw XPM image data, with the top-left corner at the given position.
- FL_EXPORT int fl_draw_pixmap (const char ∗const ∗cdata, int x, int y, Fl_Color=FL_GRAY)

  Draw XPM image data, with the top-left corner at the given position.
- FL_EXPORT int fl_draw_symbol (const char ∗label, int x, int y, int w, int h, Fl_Color)

Draw the named symbol in the given rectangle using the given color.

- void fl_end_complex_polygon ()

  Ends complex filled polygon, and draws.
- void fl_end_line ()

  Ends list of lines, and draws.
- void fl_end_loop ()

  Ends closed sequence of lines, and draws.
- void fl_end_offscreen ()

  Quit sending drawing commands to the current offscreen buffer.
- void fl_end_points ()

  Ends list of points, and draws.
- void fl_end_polygon ()

  Ends convex filled polygon, and draws.
- FL_EXPORT const char ∗ fl_expand_text (const char ∗from, char ∗buf, int maxbuf, double maxw, int &n, double &width, int wrap, int draw_symbols=0)

  Copy `from` to `buf`, replacing control characters with ^X.
- FL_EXPORT void fl_frame (const char ∗s, int x, int y, int w, int h)

  Draws a series of line segments around the given box.
- FL_EXPORT void fl_frame2 (const char ∗s, int x, int y, int w, int h)

  Draws a series of line segments around the given box.
- void fl_gap ()

  Call fl_gap() to separate loops of the path.
- void fl_line (int x, int y, int x1, int y1)

  Draws a line from (x,y) to (x1,y1)
- void fl_line (int x, int y, int x1, int y1, int x2, int y2)

  Draws a line from (x,y) to (x1,y1) and another from (x1,y1) to (x2,y2)
- void fl_line_style (int style, int width=0, char ∗dashes=0)

  Sets how to draw lines (the "pen").
- void fl_loop (int x, int y, int x1, int y1, int x2, int y2)

  Outlines a 3-sided polygon with lines.
- void fl_loop (int x, int y, int x1, int y1, int x2, int y2, int x3, int y3)

  Outlines a 4-sided polygon with lines.
- FL_EXPORT void fl_measure (const char ∗str, int &x, int &y, int draw_symbols=1)

  Measure how wide and tall the string will be when printed by the fl_draw() function with `align` parameter.
- FL_EXPORT int fl_measure_pixmap (char ∗const ∗data, int &w, int &h)

  Get the dimensions of a pixmap.
- FL_EXPORT int fl_measure_pixmap (const char ∗const ∗cdata, int &w, int &h)

  Get the dimensions of a pixmap.
- void fl_mult_matrix (double a, double b, double c, double d, double x, double y)

  Concatenates another transformation onto the current one.
- int fl_not_clipped (int x, int y, int w, int h)

  Does the rectangle intersect the current clip region?
- FL_EXPORT unsigned int fl_old_shortcut (const char ∗s)

  Emulation of XForms named shortcuts.
- FL_EXPORT void fl_overlay_clear ()

  Erase a selection rectangle without drawing a new one.
- FL_EXPORT void fl_overlay_rect (int x, int y, int w, int h)

Draws a selection rectangle, erasing a previous one by XOR'ing it first.

- void fl_pie (int x, int y, int w, int h, double a1, double a2)

  Draw filled ellipse sections using integer coordinates.
- void fl_point (int x, int y)

  Draws a single pixel at the given coordinates.
- void fl_polygon (int x, int y, int x1, int y1, int x2, int y2)

  Fills a 3-sided polygon.
- void fl_polygon (int x, int y, int x1, int y1, int x2, int y2, int x3, int y3)

  Fills a 4-sided polygon.
- void fl_pop_clip ()

  Restores the previous clip region.
- void fl_pop_matrix ()

  Restores the current transformation matrix from the stack.
- void fl_push_clip (int x, int y, int w, int h)

  Intersects the current clip region with a rectangle and pushes this new region onto the stack.
- void fl_push_matrix ()

  Saves the current transformation matrix on the stack.
- void fl_push_no_clip ()

  Pushes an empty clip region onto the stack so nothing will be clipped.
- FL_EXPORT uchar ∗ fl_read_image (uchar ∗p, int X, int Y, int W, int H, int alpha=0)

  Reads an RGB(A) image from the current window or off-screen buffer.
- void fl_rect (int x, int y, int w, int h)

  Draws a 1-pixel border inside the given bounding box.
- void fl_rect (int x, int y, int w, int h, Fl_Color c)

  Draws with passed color a 1-pixel border inside the given bounding box.
- void fl_rectf (int x, int y, int w, int h)

  Colors with current color a rectangle that exactly fills the given bounding box.
- void fl_rectf (int x, int y, int w, int h, Fl_Color c)

  Colors with passed color a rectangle that exactly fills the given bounding box.
- FL_EXPORT void fl_rectf (int x, int y, int w, int h, uchar r, uchar g, uchar b)

  Colors a rectangle with "exactly" the passed `r,g,b` color.
- FL_EXPORT void fl_reset_spot (void)
- void fl_restore_clip ()

  Undoes any clobbering of clip done by your program.
- void fl_rotate (double d)

  Concatenates rotation transformation onto the current one.
- void fl_rtl_draw (const char ∗str, int n, int x, int y)

  Draws a UTF-8 string of length `n` bytes right to left starting at the given `x, y` location.
- void fl_scale (double x, double y)

  Concatenates scaling transformation onto the current one.
- void fl_scale (double x)

  Concatenates scaling transformation onto the current one.
- FL_EXPORT void fl_scroll (int X, int Y, int W, int H, int dx, int dy, void(∗draw_area)(void ∗, int, int, int, int), void ∗data)

  Scroll a rectangle and draw the newly exposed portions.
- FL_EXPORT void fl_set_spot (int font, int size, int X, int Y, int W, int H, Fl_Window ∗win=0)
- FL_EXPORT void fl_set_status (int X, int Y, int W, int H)

- FL_EXPORT const char ∗ fl_shortcut_label (unsigned int shortcut)

  Get a human-readable string from a shortcut value.
- FL_EXPORT const char ∗ fl_shortcut_label (unsigned int shortcut, const char ∗∗eom)

  Get a human-readable string from a shortcut value.
- double fl_transform_dx (double x, double y)

  Transforms distance using current transformation matrix.
- double fl_transform_dy (double x, double y)

  Transforms distance using current transformation matrix.
- double fl_transform_x (double x, double y)

  Transforms coordinate using the current transformation matrix.
- double fl_transform_y (double x, double y)

  Transforms coordinate using the current transformation matrix.
- void fl_transformed_vertex (double xf, double yf)

  Adds coordinate pair to the vertex list without further transformations.
- void fl_translate (double x, double y)

  Concatenates translation transformation onto the current one.
- void fl_vertex (double x, double y)

  Adds a single vertex to the current path.
- void fl_xyline (int x, int y, int x1)

  Draws a horizontal line from (x,y) to (x1,y)
- void fl_xyline (int x, int y, int x1, int y2)

  Draws a horizontal line from (x,y) to (x1,y), then vertical from (x1,y) to (x1,y2)
- void fl_xyline (int x, int y, int x1, int y2, int x3)

  Draws a horizontal line from (x,y) to (x1,y), then a vertical from (x1,y) to (x1,y2) and then another horizontal from (x1,y2) to (x3,y2)
- void fl_yxline (int x, int y, int y1)

  Draws a vertical line from (x,y) to (x,y1)
- void fl_yxline (int x, int y, int y1, int x2)

  Draws a vertical line from (x,y) to (x,y1), then a horizontal from (x,y1) to (x2,y1)
- void fl_yxline (int x, int y, int y1, int x2, int y3)

  Draws a vertical line from (x,y) to (x,y1) then a horizontal from (x,y1) to (x2,y1), then another vertical from (x2,y1) to (x2,y3)

Variables

- const int **stack_max** = 16

30.7.1 Detailed Description

FLTK global graphics and GUI drawing functions. These functions are declared in <FL/fl_draw.H>, and in <FL/x.H> for offscreen buffer-related ones.

30.7.2 Macro Definition Documentation

#define fl_clip fl_push_clip

Intersects the current clip region with a rectangle and pushes this new region onto the stack (deprecated).

Parameters

`in`	*x,y,w,h*	position and size

Deprecated fl_clip(int, int, int, int) is deprecated and will be removed from future releases. Please use fl_push_clip(int x, int y, int w, int h) instead.

30.7.3 Enumeration Type Documentation

anonymous enum

Enumerator

FL_SOLID line style: ___________

FL_DASH line style: _ _ _ _ _ _

FL_DOT line style: . . . . . .

FL_DASHDOT line style: _ . _ . _ .

FL_DASHDOTDOT line style: _ . . _ . .

FL_CAP_FLAT cap style: end is flat

FL_CAP_ROUND cap style: end is round

FL_CAP_SQUARE cap style: end wraps end point

FL_JOIN_MITER join style: line join extends to a point

FL_JOIN_ROUND join style: line join is rounded

FL_JOIN_BEVEL join style: line join is tidied

30.7.4 Function Documentation

FL_EXPORT int fl_add_symbol (const char ∗ *name*, void(∗)(Fl_Color) *drawit*, int *scalable*)

Adds a symbol to the system.

Parameters

`in`	*name*	name of symbol (without the "@")
`in`	*drawit*	function to draw symbol
`in`	*scalable*	set to 1 if `drawit` uses scalable vector drawing

Returns

1 on success, 0 on failure

void fl_arc (int *x*, int *y*, int *w*, int *h*, double *a1*, double *a2*) `[inline]`

Draw ellipse sections using integer coordinates.

These functions match the rather limited circle drawing code provided by X and WIN32. The advantage over using fl_arc with floating point coordinates is that they are faster because they often use the hardware, and they draw much nicer small circles, since the small sizes are often hard-coded bitmaps.

If a complete circle is drawn it will fit inside the passed bounding box. The two angles are measured in degrees counter-clockwise from 3 o'clock and are the starting and ending angle of the arc, `a2` must be greater or equal to `a1`.

fl_arc() draws a series of lines to approximate the arc. Notice that the integer version of fl_arc() has a different number of arguments than the double version fl_arc(double x, double y, double r, double start, double end)

Parameters

in	*x,y,w,h*	bounding box of complete circle
in	*a1,a2*	start and end angles of arc measured in degrees counter-clockwise from 3 o'clock. `a2` must be greater than or equal to `a1`.

void fl_arc (double *x*, double *y*, double *r*, double *start*, double *end*) `[inline]`

Adds a series of points to the current path on the arc of a circle.

You can get elliptical paths by using scale and rotate before calling fl_arc().

Parameters

in	*x,y,r*	center and radius of circular arc
in	*start,end*	angles of start and end of arc measured in degrees counter-clockwise from 3 o'clock. If `end` is less than `start` then it draws the arc in a clockwise direction.

Examples:

```
// Draw an arc of points
fl_begin_points();
fl_arc(100.0, 100.0, 50.0, 0.0, 180.0);
fl_end_points();

// Draw arc with a line
fl_begin_line();
fl_arc(200.0, 100.0, 50.0, 0.0, 180.0);
fl_end_line();

// Draw filled arc
fl_begin_polygon();
fl_arc(300.0, 100.0, 50.0, 0.0, 180.0);
fl_end_polygon();
```

void fl_begin_complex_polygon ( ) `[inline]`

Starts drawing a complex filled polygon.

The polygon may be concave, may have holes in it, or may be several disconnected pieces. Call fl_gap() to separate loops of the path.

To outline the polygon, use fl_begin_loop() and replace each fl_gap() with fl_end_loop();fl_begin_loop() pairs.

Note

> For portability, you should only draw polygons that appear the same whether "even/odd" or "non-zero" winding rules are used to fill them. Holes should be drawn in the opposite direction to the outside loop.

void fl_begin_offscreen (Fl_Offscreen *ctx*)

Send all subsequent drawing commands to this offscreen buffer.

Parameters

ctx	the offscreen buffer.

void fl_begin_points ( ) `[inline]`

Starts drawing a list of points.

Points are added to the list with fl_vertex()

FL_EXPORT char fl_can_do_alpha_blending ( )

Checks whether platform supports true alpha blending for RGBA images.

Returns

1 if true alpha blending supported by platform
0 not supported so FLTK will use screen door transparency

void fl_circle (double *x*, double *y*, double *r*) `[inline]`

fl_circle() is equivalent to fl_arc(x,y,r,0,360), but may be faster.

It must be the *only* thing in the path: if you want a circle as part of a complex polygon you must use fl_arc()

Parameters

`in`	*x,y,r*	center and radius of circle

int fl_clip_box (int *x*, int *y*, int *w*, int *h*, int & *X*, int & *Y*, int & *W*, int & *H*) `[inline]`

Intersects the rectangle with the current clip region and returns the bounding box of the result.

Returns non-zero if the resulting rectangle is different to the original. This can be used to limit the necessary drawing to a rectangle. `W` and `H` are set to zero if the rectangle is completely outside the region.

Parameters

`in`	*x,y,w,h*	position and size of rectangle
`out`	*X,Y,W,H*	position and size of resulting bounding box.

Returns

Non-zero if the resulting rectangle is different to the original.

void fl_clip_region (Fl_Region *r*) `[inline]`

Replaces the top of the clipping stack with a clipping region of any shape.

Fl_Region is an operating system specific type.

Parameters

`in`	*r*	clipping region

void fl_copy_offscreen (int *x*, int *y*, int *w*, int *h*, Fl_Offscreen *pixmap*, int *srcx*, int *srcy*)

Copy a rectangular area of the given offscreen buffer into the current drawing destination.

Parameters

x,y	position where to draw the copied rectangle
w,h	size of the copied rectangle
pixmap	offscreen buffer containing the rectangle to copy
srcx,srcy	origin in offscreen buffer of rectangle to copy

Fl_Offscreen fl_create_offscreen (int *w*, int *h*)

Creation of an offscreen graphics buffer.

Parameters

w,h	width and height in pixels of the buffer.

Returns

the created graphics buffer.

FL_EXPORT void fl_cursor (Fl_Cursor *c*)

Sets the cursor for the current window to the specified shape and colors.

The cursors are defined in the <FL/Enumerations.H> header file.

void fl_curve (double *X0*, double *Y0*, double *X1*, double *Y1*, double *X2*, double *Y2*, double *X3*, double *Y3*) `[inline]`

Adds a series of points on a Bezier curve to the path.

The curve ends (and two of the points) are at X0,Y0 and X3,Y3.

Parameters

`in`	*X0,Y0*	curve start point
`in`	*X1,Y1*	curve control point
`in`	*X2,Y2*	curve control point
`in`	*X3,Y3*	curve end point

void fl_delete_offscreen (Fl_Offscreen *ctx*)

Deletion of an offscreen graphics buffer.

Parameters

ctx	the buffer to be deleted.

FL_EXPORT void fl_draw (const char ∗ *str*, int *x*, int *y*)

Draws a nul-terminated UTF-8 string starting at the given `x`, `y` location.

Text is aligned to the left and to the baseline of the font. To align to the bottom, subtract fl_descent() from `y`. To align to the top, subtract fl_descent() and add fl_height(). This version of fl_draw provides direct access to the text drawing function of the underlying OS. It does not apply any special handling to control characters.

FL_EXPORT void fl_draw (int *angle*, const char ∗ *str*, int *x*, int *y*)

Draws a nul-terminated UTF-8 string starting at the given `x`, `y` location and rotating `angle` degrees counter-clockwise.

This version of fl_draw provides direct access to the text drawing function of the underlying OS and is supported by Xft, Win32 and MacOS fltk subsets.

void fl_draw (int *angle*, const char ∗ *str*, int *n*, int *x*, int *y*) `[inline]`

Draws at the given `x`, `y` location a UTF-8 string of length `n` bytes rotating `angle` degrees counter-clockwise.

Note

> When using X11 (Unix, Linux, Cygwin et al.) this needs Xft to work. Under plain X11 (w/o Xft) rotated text is not supported by FLTK. A warning will be issued to stderr at runtime (only once) if you use this method with an angle other than 0.

FL_EXPORT void fl_draw (const char ∗ *str*, int *x*, int *y*, int *w*, int *h*, Fl_Align *align*, Fl_Image ∗ *img*, int *draw_symbols*)

Fancy string drawing function which is used to draw all the labels.

The string is formatted and aligned inside the passed box. Handles '\t' and '\n', expands all other control characters to '^X', and aligns inside or against the edges of the box. See Fl_Widget::align() for values of `align`. The value FL_ALIGN_INSIDE is ignored, as this function always prints inside the box. If `img` is provided and is not `NULL`, the image is drawn above or below the text as specified by the `align` value. The `draw_symbols` argument specifies whether or not to look for symbol names starting with the '@' character'

FL_EXPORT void fl_draw_box (Fl_Boxtype *t*, int *x*, int *y*, int *w*, int *h*, Fl_Color *c*)

Draws a box using given type, position, size and color.

Parameters

`in`	*t*	box type
`in`	*x,y,w,h*	position and size
`in`	*c*	color

void fl_draw_image (const uchar ∗ *buf*, int *X*, int *Y*, int *W*, int *H*, int *D* = *3*, int *L* = *0*) `[inline]`

Draws an 8-bit per color RGB or luminance image.

Parameters

`in`	*buf*	points at the "r" data of the top-left pixel. Color data must be in `r,g,b` order. Luminance data is only one `gray` byte.
`in`	*X,Y*	position where to put top-left corner of image
`in`	*W,H*	size of the image
`in`	*D*	delta to add to the pointer between pixels. It may be any value greater than or equal to 1, or it can be negative to flip the image horizontally
`in`	*L*	delta to add to the pointer between lines (if 0 is passed it uses `W ∗ D`), and may be larger than `W ∗ D` to crop data, or negative to flip the image vertically

It is highly recommended that you put the following code before the first `show()` of *any* window in your program to get rid of the dithering if possible:

```
Fl::visual(FL_RGB);
```

Gray scale (1-channel) images may be drawn. This is done if `abs(D)` is less than 3, or by calling fl_draw_image_mono(). Only one 8-bit sample is used for each pixel, and on screens with different numbers of bits for red, green, and blue only gray colors are used. Setting `D` greater than 1 will let you display one channel of a color image.

Note:

> The X version does not support all possible visuals. If FLTK cannot draw the image in the current visual it will abort. FLTK supports any visual of 8 bits or less, and all common TrueColor visuals up to 32 bits.

void fl_draw_image (Fl_Draw_Image_Cb *cb*, void ∗ *data*, int *X*, int *Y*, int *W*, int *H*, int *D* = *3*) `[inline]`

Draws an image using a callback function to generate image data.

You can generate the image as it is being drawn, or do arbitrary decompression of stored data, provided it can be decompressed to individual scan lines easily.

Parameters

`in`	*cb*	callback function to generate scan line data
`in`	*data*	user data passed to callback function
`in`	*X,Y*	screen position of top left pixel
`in`	*W,H*	image width and height
`in`	*D*	data size in bytes (must be greater than 0)

See Also

fl_draw_image(const uchar∗ buf, int X,int Y,int W,int H, int D, int L)

The callback function `cb` is called with the `void* data` user data pointer to allow access to a structure of information about the image, and the `x`, `y`, and `w` of the scan line desired from the image. 0,0 is the upper-left corner of the image, not `x`, `y`. A pointer to a buffer to put the data into is passed. You must copy `w` pixels from scanline `y`, starting at pixel `x`, to this buffer.

Due to cropping, less than the whole image may be requested. So `x` may be greater than zero, the first `y` may be greater than zero, and `w` may be less than `W`. The buffer is long enough to store the entire `W * D` pixels, this is for convenience with some decompression schemes where you must decompress the entire line at once: decompress it into the buffer, and then if `x` is not zero, copy the data over so the `x'th` pixel is at the start of the buffer.

You can assume the `y's` will be consecutive, except the first one may be greater than zero.

If `D` is 4 or more, you must fill in the unused bytes with zero.

void fl_draw_image_mono (const uchar ∗ *buf*, int *X*, int *Y*, int *W*, int *H*, int *D* = *1*, int *L* = *0*) `[inline]`

Draws a gray-scale (1 channel) image.

See Also

fl_draw_image(const uchar∗ buf, int X,int Y,int W,int H, int D, int L)

void fl_draw_image_mono (Fl_Draw_Image_Cb *cb*, void ∗ *data*, int *X*, int *Y*, int *W*, int *H*, int *D* = *1*) `[inline]`

Draws a gray-scale image using a callback function to generate image data.

See Also

fl_draw_image(Fl_Draw_Image_Cb cb, void∗ data, int X,int Y,int W,int H, int D)

FL_EXPORT int fl_draw_pixmap (char ∗const ∗ *data*, int *x*, int *y*, Fl_Color *bg*)

Draw XPM image data, with the top-left corner at the given position.

The image is dithered on 8-bit displays so you won't lose color space for programs displaying both images and pixmaps.

Parameters

`in`	*data*	pointer to XPM image data
`in`	*x,y*	position of top-left corner
`in`	*bg*	background color

Returns

0 if there was any error decoding the XPM data.

FL_EXPORT int fl_draw_pixmap (const char ∗const ∗ *cdata*, int *x*, int *y*, Fl_Color *bg*)

Draw XPM image data, with the top-left corner at the given position.

See Also

fl_draw_pixmap(char∗ const∗ data, int x, int y, Fl_Color bg)

FL_EXPORT int fl_draw_symbol (const char ∗ *label*, int *x*, int *y*, int *w*, int *h*, Fl_Color *col*)

Draw the named symbol in the given rectangle using the given color.
Parameters

`in`	*label*	name of symbol
`in`	*x,y*	position of symbol
`in`	*w,h*	size of symbol
`in`	*col*	color of symbox

Returns

1 on success, 0 on failure

FL_EXPORT const char∗ fl_expand_text (const char ∗ *from*, char ∗ *buf*, int *maxbuf*, double *maxw*, int & *n*, double & *width*, int *wrap*, int *draw_symbols*)

Copy `from` to `buf`, replacing control characters with ^X.

Stop at a newline or if `maxbuf` characters written to buffer. Also word-wrap if width exceeds maxw. Returns a pointer to the start of the next line of characters. Sets n to the number of characters put into the buffer. Sets width to the width of the string in the current font.

FL_EXPORT void fl_frame (const char ∗ *s*, int *x*, int *y*, int *w*, int *h*)

Draws a series of line segments around the given box.

The string `s` must contain groups of 4 letters which specify one of 24 standard grayscale values, where 'A' is black and 'X' is white. The order of each set of 4 characters is: top, left, bottom, right. The result of calling fl_frame() with a string that is not a multiple of 4 characters in length is undefined. The only difference between this function and fl_frame2() is the order of the line segments.
Parameters

`in`	*s*	sets of 4 grayscale values in top, left, bottom, right order

`in`	*x,y,w,h*	position and size

FL_EXPORT void fl_frame2 (const char ∗ *s*, int *x*, int *y*, int *w*, int *h*)

Draws a series of line segments around the given box.

The string `s` must contain groups of 4 letters which specify one of 24 standard grayscale values, where 'A' is black and 'X' is white. The order of each set of 4 characters is: bottom, right, top, left. The result of calling fl_frame2() with a string that is not a multiple of 4 characters in length is undefined. The only difference between this function and fl_frame() is the order of the line segments.

Parameters

`in`	*s*	sets of 4 grayscale values in bottom, right, top, left order
`in`	*x,y,w,h*	position and size

void fl_gap ( ) `[inline]`

Call fl_gap() to separate loops of the path.

It is unnecessary but harmless to call fl_gap() before the first vertex, after the last vertex, or several times in a row.

void fl_line_style (int *style*, int *width* = *0*, char ∗ *dashes* = *0*) `[inline]`

Sets how to draw lines (the "pen").

If you change this it is your responsibility to set it back to the default using `fl_line_style(0)`.

Parameters

`in`	*style*	A bitmask which is a bitwise-OR of a line style, a cap style, and a join style. If you don't specify a dash type you will get a solid line. If you don't specify a cap or join type you will get a system-defined default of whatever value is fastest.
`in`	*width*	The thickness of the lines in pixels. Zero results in the system defined default, which on both X and Windows is somewhat different and nicer than 1.
`in`	*dashes*	A pointer to an array of dash lengths, measured in pixels. The first location is how long to draw a solid portion, the next is how long to draw the gap, then the solid, etc. It is terminated with a zero-length entry. A `NULL` pointer or a zero-length array results in a solid line. Odd array sizes are not supported and result in undefined behavior.

Note

Because of how line styles are implemented on Win32 systems, you *must* set the line style *after* setting the drawing color. If you set the color after the line style you will lose the line style settings. The `dashes` array does not work under Windows 95, 98 or Me, since those operating systems do not support complex line styles.

FL_EXPORT void fl_measure (const char ∗ *str*, int & *w*, int & *h*, int *draw_symbols*)

Measure how wide and tall the string will be when printed by the fl_draw() function with `align` parameter.

If the incoming `w` is non-zero it will wrap to that width.

The current font is used to do the width/height calculations, so unless its value is known at the time fl_measure() is called, it is advised to first set the current font with fl_font(). With event-driven GUI programming you can never be sure which widget was exposed and redrawn last, nor which font it used. If you have not called fl_font() explicitly in your own code, the width and height may be set to unexpected values, even zero!

Note: In the general use case, it's a common error to forget to set `w` to 0 before calling fl_measure() when wrap behavior isn't needed.

Parameters

in	*str*	nul-terminated string
out	*w,h*	width and height of string in current font
in	*draw_symbols*	non-zero to enable @symbol handling [default=1]

```
// Example: Common use case for fl_measure()
const char *s = "This is a test";
int wi=0, hi=0;              // initialize to zero before calling fl_measure()
fl_font(FL_HELVETICA, 14);   // set current font face/size to be used for measuring
fl_measure(s, wi, hi);       // returns pixel width/height of string in current font
```

FL_EXPORT int fl_measure_pixmap (char ∗const ∗ *data,* int & *w,* int & *h*)

Get the dimensions of a pixmap.

An XPM image contains the dimensions in its data. This function returns the width and height.

Parameters

in	*data*	pointer to XPM image data.
out	*w,h*	width and height of image

Returns

non-zero if the dimensions were parsed OK
0 if there were any problems

FL_EXPORT int fl_measure_pixmap (const char ∗const ∗ *cdata,* int & *w,* int & *h*)

Get the dimensions of a pixmap.

See Also

fl_measure_pixmap(char∗ const∗ data, int &w, int &h)

void fl_mult_matrix (double *a,* double *b,* double *c,* double *d,* double *x,* double *y*) `[inline]`

Concatenates another transformation onto the current one.

Parameters

in	*a,b,c,d,x,y*	transformation matrix elements such that `X' = aX + cY + x` and `Y' = bX +dY + y`

int fl_not_clipped (int *x,* int *y,* int *w,* int *h*) `[inline]`

Does the rectangle intersect the current clip region?

Parameters

in	*x,y,w,h*	position and size of rectangle

Returns

non-zero if any of the rectangle intersects the current clip region. If this returns 0 you don't have to draw the object.

Note

Under X this returns 2 if the rectangle is partially clipped, and 1 if it is entirely inside the clip region.

FL_EXPORT unsigned int fl_old_shortcut (const char ∗ *s*)

Emulation of XForms named shortcuts.

Converts ascii shortcut specifications (eg. "^c") into the FLTK integer equivalent (eg. FL_CTRL+'c')
These ascii characters are used to specify the various keyboard modifier keys:

```
# - Alt
+ - Shift
^ - Control
! - Meta
@ - Command (Ctrl on linux/win, Meta on OSX)
```

These special characters can be combined to form chords of modifier keys. (See 'Remarks' below)

After the optional modifier key prefixes listed above, one can either specify a single keyboard character to use as the shortcut, or a numeric sequence in hex, decimal or octal.

Examples:

```
"c"       -- Uses 'c' as the shortcut
"#^c"     -- Same as FL_ALT|FL_CTRL|'c'
"#^!c"    -- Same as FL_ALT|FL_CTRL|FL_META|'c'
"@c"      -- Same as FL_COMMAND|'c' (see FL_COMMAND for platform specific behavior)
"0x63"    -- Same as "c" (hex 63=='c')
"99"      -- Same as "c" (dec 99=='c')
"0143"    -- Same as "c" (octal 0143=='c')
"^0x63"   -- Same as (FL_CTRL|'c'), or (FL_CTRL|0x63)
"^99"     -- Same as (FL_CTRL|'c'), or (FL_CTRL|99)
"^0143"   -- Same as (FL_CTRL|'c'), or (FL_CTRL|0143)
```

Remarks

Due to XForms legacy, there are some odd things to consider when using the modifier characters.
(1) You can use the special modifier keys for chords *only* if the modifiers are provided in this order: #, +, ^, !, @. Other ordering can yield undefined results.
So for instance, Ctrl-Alt-c must be specified as "#^c" (and not "^#c"), due to the above ordering rule.
(2) If you want to make a shortcut that uses one of the special modifier characters (as the character being modified), then to avoid confusion, specify the numeric equivalent, e.g.

```
If you want..                    Then use..
-----------------------------    ------------------------------
'#' as the shortcut..            "0x23"  (instead of just "#").
'+' as the shortcut..            "0x2b"  (instead of just "+").
'^' as the shortcut..            "0x5e"  (instead of just "^").
Alt-+ as the shortcut..          "#0x2b" (instead of "#+").
Alt-^ as the shortcut..          "#0x5e" (instead of "#^").
..etc..
```

As a general rule that's easy to remember, unless the shortcut key to be modified is a single alphanumeric character [A-Z,a-z,0-9), it's probably best to use the numeric equivalents.

Todo Fix these silly legacy issues in a future release to support more predictable behavior for the modifier keys.

void fl_pie (int *x*, int *y*, int *w*, int *h*, double *a1*, double *a2*) `[inline]`

Draw filled ellipse sections using integer coordinates.

Like fl_arc(), but fl_pie() draws a filled-in pie slice. This slice may extend outside the line drawn by fl_arc(); to avoid this use w - 1 and h - 1.

Parameters

`in`	*x,y,w,h*	bounding box of complete circle
`in`	*a1,a2*	start and end angles of arc measured in degrees counter-clockwise from 3 o'clock. `a2` must be greater than or equal to `a1`.

void fl_polygon (int *x*, int *y*, int *x1*, int *y1*, int *x2*, int *y2*) `[inline]`

Fills a 3-sided polygon.

The polygon must be convex.

void fl_polygon (int *x*, int *y*, int *x1*, int *y1*, int *x2*, int *y2*, int *x3*, int *y3*) `[inline]`

Fills a 4-sided polygon.

The polygon must be convex.

void fl_pop_clip ( ) `[inline]`

Restores the previous clip region.

You must call fl_pop_clip() once for every time you call fl_push_clip(). Unpredictable results may occur if the clip stack is not empty when you return to FLTK.

void fl_push_clip (int *x*, int *y*, int *w*, int *h*) `[inline]`

Intersects the current clip region with a rectangle and pushes this new region onto the stack.

Parameters

`in`	*x,y,w,h*	position and size

void fl_push_matrix ( ) `[inline]`

Saves the current transformation matrix on the stack.

The maximum depth of the stack is 32.

FL_EXPORT uchar∗ fl_read_image (uchar ∗ *p*, int *X*, int *Y*, int *W*, int *H*, int *alpha* = `0`)

Reads an RGB(A) image from the current window or off-screen buffer.

Parameters

`in`	*p*	pixel buffer, or NULL to allocate one
`in`	*X,Y*	position of top-left of image to read
`in`	*W,H*	width and height of image to read
`in`	*alpha*	alpha value for image (0 for none)

Returns

pointer to pixel buffer, or NULL if allocation failed.

The `p` argument points to a buffer that can hold the image and must be at least `W*H*3` bytes when reading RGB images, or `W*H*4` bytes when reading RGBA images. If NULL, fl_read_image() will create an array of the proper size which can be freed using `delete[]`.

The `alpha` parameter controls whether an alpha channel is created and the value that is placed in the alpha channel. If 0, no alpha channel is generated.

void fl_rect (int *x*, int *y*, int *w*, int *h*) `[inline]`

Draws a 1-pixel border *inside* the given bounding box.

This function is meant for quick drawing of simple boxes. The behavior is undefined for line widths that are not 1.

FL_EXPORT void fl_rectf (int *x*, int *y*, int *w*, int *h*, uchar *r*, uchar *g*, uchar *b*)

Colors a rectangle with "exactly" the passed `r,g,b` color.

On screens with less than 24 bits of color this is done by drawing a solid-colored block using fl_draw_-image() so that the correct color shade is produced.

FL_EXPORT void fl_reset_spot (void)

Todo provide user documentation for fl_reset_spot function

void fl_rotate (double *d*) `[inline]`

Concatenates rotation transformation onto the current one.
Parameters

`in`	*d*	- rotation angle, counter-clockwise in degrees (not radians)

void fl_scale (double *x*, double *y*) `[inline]`

Concatenates scaling transformation onto the current one.
Parameters

`in`	*x,y*	scale factors in x-direction and y-direction

void fl_scale (double *x*) `[inline]`

Concatenates scaling transformation onto the current one.
Parameters

`in`	*x*	scale factor in both x-direction and y-direction

FL_EXPORT void fl_scroll (int *X*, int *Y*, int *W*, int *H*, int *dx*, int *dy*, void(*)(void *, int, int, int, int) *draw_area*, void * *data*)

Scroll a rectangle and draw the newly exposed portions.

Parameters

in	*X,Y*	position of top-left of rectangle
in	*W,H*	size of rectangle
in	*dx,dy*	pixel offsets for shifting rectangle
in	*draw_area*	callback function to draw rectangular areas
in	*data*	pointer to user data for callback The contents of the rectangular area is first shifted by `dx` and `dy` pixels. The `draw_area` callback is then called for every newly exposed rectangular area.

FL_EXPORT void fl_set_spot (int *font*, int *size*, int *X*, int *Y*, int *W*, int *H*, Fl_Window ∗ *win* = `0`)

Todo provide user documentation for fl_set_spot function

FL_EXPORT void fl_set_status (int *X*, int *Y*, int *W*, int *H*)

Todo provide user documentation for fl_set_status function

FL_EXPORT const char∗ fl_shortcut_label (unsigned int *shortcut*)

Get a human-readable string from a shortcut value.

Unparse a shortcut value as used by Fl_Button or Fl_Menu_Item into a human-readable string like "-Alt+N". This only works if the shortcut is a character key or a numbered function key. If the shortcut is zero then an empty string is returned. The return value points at a static buffer that is overwritten with each call.

Since

FLTK 1.3.4 modifier key names can be localized, but key names can not yet be localized. This may be added to a future FLTK version.

Modifier key names (human-readable shortcut names) can be defined with the following global const char ∗ pointer variables:

- fl_local_ctrl => name of FL_CTRL
- fl_local_alt => name of FL_ALT
- fl_local_shift => name of FL_SHIFT
- fl_local_meta => name of FL_META

```
fl_local_ctrl = "Strg";      // German for "Ctrl"
fl_local_shift = "Umschalt"; // German for "Shift"
```

The shortcut name will be constructed by adding all modifier names in the order defined above plus the name of the key. A '+' character is added to each modifier name unless it has a trailing '\' or a trailing '+'.

Example:

Ctrl+Alt+Shift+Meta+F12

The default values for modifier key names are as given above for all platforms except Mac OS X. Mac OS X uses graphical characters that represent the typical OS X modifier names in menus, e.g. cloverleaf, saucepan, etc. You may, however, redefine Mac OS X modifier names as well.

Parameters

in	*shortcut*	the integer value containing the ascii character or extended keystroke plus modifiers

Returns

a pointer to a static buffer containing human readable text for the shortcut

FL_EXPORT const char∗ fl_shortcut_label (unsigned int *shortcut*, const char ∗∗ *eom*)

Get a human-readable string from a shortcut value.
Parameters

in	*shortcut*	the integer value containing the ascii character or extended keystroke plus modifiers
in	*eom*	if this pointer is set, it will receive a pointer to the end of the modifier text

Returns

a pointer to a static buffer containing human readable text for the shortcut

See Also

fl_shortcut_label(unsigned int shortcut)

double fl_transform_dx (double *x*, double *y*) `[inline]`

Transforms distance using current transformation matrix.
Parameters

in	*x,y*	coordinate

double fl_transform_dy (double *x*, double *y*) `[inline]`

Transforms distance using current transformation matrix.
Parameters

in	*x,y*	coordinate

double fl_transform_x (double *x*, double *y*) `[inline]`

Transforms coordinate using the current transformation matrix.
Parameters

in	*x,y*	coordinate

double fl_transform_y (double *x*, double *y*) `[inline]`

Transforms coordinate using the current transformation matrix.

Parameters

`in`	*x,y*	coordinate

void fl_transformed_vertex (double *xf*, double *yf*) `[inline]`

Adds coordinate pair to the vertex list without further transformations.
Parameters

`in`	*xf,yf*	transformed coordinate

void fl_translate (double *x*, double *y*) `[inline]`

Concatenates translation transformation onto the current one.
Parameters

`in`	*x,y*	translation factor in x-direction and y-direction

void fl_vertex (double *x*, double *y*) `[inline]`

Adds a single vertex to the current path.
Parameters

`in`	*x,y*	coordinate

30.8 Multithreading support functions

fl multithreading support functions declared in <FL/Fl.H>

Functions

- static void Fl::awake (void ∗message=0)

  Sends a message pointer to the main thread, causing any pending Fl::wait() call to terminate so that the main thread can retrieve the message and any pending redraws can be processed.
- static int Fl::awake (Fl_Awake_Handler cb, void ∗message=0)

  See void awake(void∗ message=0).
- static int Fl::lock ()

  The lock() method blocks the current thread until it can safely access FLTK widgets and data.
- static void ∗ Fl::thread_message ()

  The thread_message() method returns the last message that was sent from a child by the awake() method.
- static void Fl::unlock ()

  The unlock() method releases the lock that was set using the lock() method.

30.8.1 Detailed Description

fl multithreading support functions declared in <FL/Fl.H>

30.8.2 Function Documentation

void Fl::awake (void ∗ *msg* = *0*) `[static]`

Sends a message pointer to the main thread, causing any pending Fl::wait() call to terminate so that the main thread can retrieve the message and any pending redraws can be processed.

Multiple calls to Fl::awake() will queue multiple pointers for the main thread to process, up to a system-defined (typically several thousand) depth. The default message handler saves the last message which can be accessed using the Fl::thread_message() function.

In the context of a threaded application, a call to Fl::awake() with no argument will trigger event loop handling in the main thread. Since it is not possible to call Fl::flush() from a subsidiary thread, Fl::awake() is the best (and only, really) substitute.

See also: Multithreading

int Fl::awake (Fl_Awake_Handler *func*, void ∗ *data* = *0*) `[static]`

See void awake(void∗ message=0).

Let the main thread know an update is pending and have it call a specific function.

Registers a function that will be called by the main thread during the next message handling cycle. Returns 0 if the callback function was registered, and -1 if registration failed. Over a thousand awake callbacks can be registered simultaneously.

See Also

Fl::awake(void∗ message=0)

int Fl::lock ( ) `[static]`

The lock() method blocks the current thread until it can safely access FLTK widgets and data.

Child threads should call this method prior to updating any widgets or accessing data. The main thread must call lock() to initialize the threading support in FLTK. lock() will return non-zero if threading is not available on the platform.

Child threads must call unlock() when they are done accessing FLTK.

When the wait() method is waiting for input or timeouts, child threads are given access to FLTK. Similarly, when the main thread needs to do processing, it will wait until all child threads have called unlock() before processing additional data.

Returns

0 if threading is available on the platform; non-zero otherwise.

See also: Multithreading

void $*$ Fl::thread_message ( ) `[static]`

The thread_message() method returns the last message that was sent from a child by the awake() method.

See also: Multithreading

void Fl::unlock ( ) `[static]`

The unlock() method releases the lock that was set using the lock() method.

Child threads should call this method as soon as they are finished accessing FLTK.

See also: Multithreading

30.9 Safe widget deletion support functions

These functions, declared in <FL/Fl.H>, support deletion of widgets inside callbacks.

Functions

- static void Fl::clear_widget_pointer (Fl_Widget const ∗w)

  Clears a widget pointer in the watch list.
- static void Fl::delete_widget (Fl_Widget ∗w)

  Schedules a widget for deletion at the next call to the event loop.
- static void Fl::do_widget_deletion ()

  Deletes widgets previously scheduled for deletion.
- static void Fl::release_widget_pointer (Fl_Widget ∗&w)

  Releases a widget pointer from the watch list.
- static void Fl::watch_widget_pointer (Fl_Widget ∗&w)

  Adds a widget pointer to the widget watch list.

30.9.1 Detailed Description

These functions, declared in <FL/Fl.H>, support deletion of widgets inside callbacks. Fl::delete_widget() should be called when deleting widgets or complete widget trees (Fl_Group, Fl_Window, ...) inside callbacks.

The other functions are intended for internal use. The preferred way to use them is by using the helper class Fl_Widget_Tracker.

The following is to show how it works ...

There are three groups of related methods:

1. scheduled widget deletion
   - Fl::delete_widget() schedules widgets for deletion
   - Fl::do_widget_deletion() deletes all scheduled widgets
2. widget watch list (”smart pointers”)
   - Fl::watch_widget_pointer() adds a widget pointer to the watch list
   - Fl::release_widget_pointer() removes a widget pointer from the watch list
   - Fl::clear_widget_pointer() clears a widget pointer *in* the watch list
3. the class Fl_Widget_Tracker:
   - the constructor calls Fl::watch_widget_pointer()
   - the destructor calls Fl::release_widget_pointer()
   - the access methods can be used to test, if a widget has been deleted

     See Also

     Fl_Widget_Tracker.

30.9.2 Function Documentation

void Fl::clear_widget_pointer (Fl_Widget const ∗ *w*) `[static]`

Clears a widget pointer *in* the watch list.

This is called when a widget is destroyed (by its destructor). You should never call this directly.

Note

Internal use only !

This method searches the widget watch list for pointers to the widget and clears each pointer that points to it. Widget pointers can be added to the widget watch list by calling Fl::watch_widget_pointer() or by using the helper class Fl_Widget_Tracker (recommended).

See Also

Fl::watch_widget_pointer()
class Fl_Widget_Tracker

void Fl::delete_widget (Fl_Widget ∗ *wi*) `[static]`

Schedules a widget for deletion at the next call to the event loop.

Use this method to delete a widget inside a callback function.

To avoid early deletion of widgets, this function should be called toward the end of a callback and only after any call to the event loop (Fl::wait(), Fl::flush(), Fl::check(), fl_ask(), etc.).

When deleting groups or windows, you must only delete the group or window widget and not the individual child widgets.

Since

FLTK 1.3.4 the widget will be hidden immediately, but the actual destruction will be delayed until the event loop is finished. Up to FLTK 1.3.3 windows wouldn't be hidden before the event loop was done, hence you had to hide() a window in your window close callback if you called Fl::delete_widget() to destroy (and hide) the window.
FLTK 1.3.0 it is not necessary to remove widgets from their parent groups or windows before calling this, because it will be done in the widget's destructor, but it is not a failure to do this nevertheless.

Note

In FLTK 1.1 you **must** remove widgets from their parent group (or window) before deleting them.

See Also

Fl_Widget::∼Fl_Widget()

void Fl::do_widget_deletion ( ) `[static]`

Deletes widgets previously scheduled for deletion.

This is for internal use only. You should never call this directly.

Fl::do_widget_deletion() is called from the FLTK event loop or whenever you call Fl::wait(). The previously scheduled widgets are deleted in the same order they were scheduled by calling Fl::delete_widget().

See Also

Fl::delete_widget(Fl_Widget ∗wi)

void Fl::release_widget_pointer (Fl_Widget *& w) `[static]`

Releases a widget pointer from the watch list.

This is used to remove a widget pointer that has been added to the watch list with Fl::watch_widget_-pointer(), when it is not needed anymore.

Note

Internal use only, please use class Fl_Widget_Tracker instead.

See Also

Fl::watch_widget_pointer()

void Fl::watch_widget_pointer (Fl_Widget *& w) `[static]`

Adds a widget pointer to the widget watch list.

Note

Internal use only, please use class Fl_Widget_Tracker instead.

This can be used, if it is possible that a widget might be deleted during a callback or similar function. The widget pointer must be added to the watch list before calling the callback. After the callback the widget pointer can be queried, if it is NULL. *If* it is NULL, then the widget has been deleted during the callback and must not be accessed anymore. If the widget pointer is *not* NULL, then the widget has not been deleted and can be accessed safely.

After accessing the widget, the widget pointer must be released from the watch list by calling Fl-::release_widget_pointer().

Example for a button that is clicked (from its handle() method):

```
Fl_Widget *wp = this;           // save 'this' in a pointer variable
Fl::watch_widget_pointer(wp); // add the pointer to the watch list
set_changed();                        // set the changed flag
do_callback();                        // call the callback
if (!wp) {                            // the widget has been deleted

  // DO NOT ACCESS THE DELETED WIDGET !

} else {                              // the widget still exists
  clear_changed();                    // reset the changed flag
}

Fl::release_widget_pointer(wp);       // remove the pointer from the watch list
```

This works, because all widgets call Fl::clear_widget_pointer() in their destructors.

See Also

Fl::release_widget_pointer()
Fl::clear_widget_pointer()

An easier and more convenient method to control widget deletion during callbacks is to use the class Fl_-Widget_Tracker with a local (automatic) variable.

See Also

class Fl_Widget_Tracker

30.10 Cairo Support Functions and Classes

Classes

- class Fl_Cairo_State

  Contains all the necessary info on the current cairo context.
- class Fl_Cairo_Window

  This defines a pre-configured cairo fltk window.

Functions

- static void Fl::cairo_autolink_context (bool alink)

  when FLTK_HAVE_CAIRO is defined and cairo_autolink_context() is true, any current window dc is linked to a current cairo context.
- static bool Fl::cairo_autolink_context ()

  Gets the current autolink mode for cairo support.
- static cairo_t ∗ Fl::cairo_cc ()

  Gets the current cairo context linked with a fltk window.
- static void Fl::cairo_cc (cairo_t ∗c, bool own=false)

  Sets the current cairo context to `c`*.*
- static cairo_t ∗ Fl::cairo_make_current (Fl_Window ∗w)

  Provides a corresponding cairo context for window wi.

30.10.1 Detailed Description

30.10.2 Function Documentation

static void Fl::cairo_autolink_context (bool *alink*) `[inline], [static]`

when FLTK_HAVE_CAIRO is defined and cairo_autolink_context() is true, any current window dc is linked to a current cairo context.

This is not the default, because it may not be necessary to add cairo support to all fltk supported windows. When you wish to associate a cairo context in this mode, you need to call explicitly in your draw() overridden method, Fl::cairo_make_current(Fl_Window∗). This will create a cairo context but only for this Window. Still in custom cairo application it is possible to handle completely this process automatically by setting `alink` to true. In this last case, you don't need anymore to call Fl::cairo_make_current(). You can use Fl::cairo_cc() to get the current cairo context anytime.

Note

Only available when configure has the –enable-cairo option

static bool Fl::cairo_autolink_context ( ) `[inline], [static]`

Gets the current autolink mode for cairo support.
Return values

false	if no cairo context autolink is made for each window.
true	if any fltk window is attached a cairo context when it is current.

See Also

void cairo_autolink_context(bool alink)

Note

Only available when configure has the –enable-cairo option

static cairo_t* Fl::cairo_cc ( ) `[inline]`, `[static]`

Gets the current cairo context linked with a fltk window.

static void Fl::cairo_cc (cairo_t * *c*, bool *own* = *`false`*) `[inline]`, `[static]`

Sets the current cairo context to `c`.

Set `own` to true if you want fltk to handle this cc deletion.

Note

Only available when configure has the –enable-cairo option

cairo_t * Fl::cairo_make_current (Fl_Window * *wi*) `[static]`

Provides a corresponding cairo context for window *wi*.

This is needed in a draw() override if Fl::cairo_autolink_context() returns false, which is the default. The cairo_context() does not need to be freed as it is freed every time a new cairo context is created. When the program terminates, a call to Fl::cairo_make_current(0) will destroy any residual context.

Note

A new cairo context is not always re-created when this method is used. In particular, if the current graphical context and the current window didn't change between two calls, the previous gc is internally kept, thus optimizing the drawing performances. Also, after this call, Fl::cairo_cc() is adequately updated with this cairo context.
Only available when configure has the –enable-cairo option

Returns

the valid cairo_t* cairo context associated to this window.

30.11 Unicode and UTF-8 functions

fl global Unicode and UTF-8 handling functions declared in <FL/fl_utf8.h>

Macros

- #define ERRORS_TO_CP1252 1
- #define ERRORS_TO_ISO8859_1 1
- #define **NBC** 0xFFFF + 1
- #define STRICT_RFC3629 0

Functions

- FL_EXPORT int fl_access (const char ∗f, int mode)

  Cross-platform function to test a files access() with a UTF-8 encoded name or value.
- FL_EXPORT int fl_chmod (const char ∗f, int mode)

  Cross-platform function to set a files mode() with a UTF-8 encoded name or value.
- FL_EXPORT int **fl_execvp** (const char ∗file, char ∗const ∗argv)
- FL_EXPORT FILE ∗ fl_fopen (const char ∗f, const char ∗mode)

  Cross-platform function to open files with a UTF-8 encoded name.
- FL_EXPORT char ∗ fl_getcwd (char ∗b, int l)

  Cross-platform function to get the current working directory as a UTF-8 encoded value.
- FL_EXPORT char ∗ fl_getenv (const char ∗v)

  Cross-platform function to get environment variables with a UTF-8 encoded name or value.
- FL_EXPORT char fl_make_path (const char ∗path)

  Cross-platform function to recursively create a path in the file system.
- FL_EXPORT void fl_make_path_for_file (const char ∗path)

  Cross-platform function to create a path for the file in the file system.
- FL_EXPORT int fl_mkdir (const char ∗f, int mode)

  Cross-platform function to create a directory with a UTF-8 encoded name.
- FL_EXPORT unsigned int fl_nonspacing (unsigned int ucs)

  Returns true if the Unicode character `ucs` is non-spacing.
- FL_EXPORT int fl_open (const char ∗f, int oflags,...)

  Cross-platform function to open files with a UTF-8 encoded name.
- FL_EXPORT int fl_rename (const char ∗f, const char ∗n)

  Cross-platform function to rename a filesystem object using UTF-8 encoded names.
- FL_EXPORT int fl_rmdir (const char ∗f)

  Cross-platform function to remove a directory with a UTF-8 encoded name.
- FL_EXPORT int fl_stat (const char ∗f, struct stat ∗b)

  Cross-platform function to stat() a file using a UTF-8 encoded name or value.
- FL_EXPORT int fl_system (const char ∗cmd)

  Cross-platform function to run a system command with a UTF-8 encoded string.
- FL_EXPORT int fl_tolower (unsigned int ucs)

  Returns the Unicode lower case value of `ucs`.
- FL_EXPORT int fl_toupper (unsigned int ucs)

  Returns the Unicode upper case value of `ucs`.
- FL_EXPORT unsigned fl_ucs_to_Utf16 (const unsigned ucs, unsigned short ∗dst, const unsigned dstlen)
- FL_EXPORT int fl_unlink (const char ∗f)

Cross-platform function to unlink() (that is, delete) a file using a UTF-8 encoded filename.
- FL_EXPORT char ∗ fl_utf2mbcs (const char ∗s)

  Converts UTF-8 string `s` *to a local multi-byte character string.*
- FL_EXPORT const char ∗ fl_utf8back (const char ∗p, const char ∗start, const char ∗end)
- FL_EXPORT int fl_utf8bytes (unsigned ucs)

  Return the number of bytes needed to encode the given UCS4 character in UTF-8.
- FL_EXPORT unsigned fl_utf8decode (const char ∗p, const char ∗end, int ∗len)
- FL_EXPORT int fl_utf8encode (unsigned ucs, char ∗buf)
- FL_EXPORT unsigned fl_utf8from_mb (char ∗dst, unsigned dstlen, const char ∗src, unsigned srclen)
- FL_EXPORT unsigned fl_utf8froma (char ∗dst, unsigned dstlen, const char ∗src, unsigned srclen)
- FL_EXPORT unsigned fl_utf8fromwc (char ∗dst, unsigned dstlen, const wchar_t ∗src, unsigned srclen)
- FL_EXPORT const char ∗ fl_utf8fwd (const char ∗p, const char ∗start, const char ∗end)
- FL_EXPORT int fl_utf8len (char c)

  Returns the byte length of the UTF-8 sequence with first byte `c`*, or -1 if* `c` *is not valid.*
- FL_EXPORT int fl_utf8len1 (char c)

  Returns the byte length of the UTF-8 sequence with first byte `c`*, or 1 if* `c` *is not valid.*
- FL_EXPORT int fl_utf8locale (void)
- FL_EXPORT int fl_utf8test (const char ∗src, unsigned len)
- FL_EXPORT unsigned fl_utf8to_mb (const char ∗src, unsigned srclen, char ∗dst, unsigned dstlen)
- FL_EXPORT unsigned fl_utf8toa (const char ∗src, unsigned srclen, char ∗dst, unsigned dstlen)
- FL_EXPORT unsigned fl_utf8toUtf16 (const char ∗src, unsigned srclen, unsigned short ∗dst, unsigned dstlen)
- FL_EXPORT unsigned fl_utf8towc (const char ∗src, unsigned srclen, wchar_t ∗dst, unsigned dstlen)

  Converts a UTF-8 string into a wide character string.
- FL_EXPORT int fl_utf_nb_char (const unsigned char ∗buf, int len)

  Returns the number of Unicode chars in the UTF-8 string.
- FL_EXPORT int fl_utf_strcasecmp (const char ∗s1, const char ∗s2)

  UTF-8 aware strcasecmp - converts to Unicode and tests.
- FL_EXPORT int fl_utf_strncasecmp (const char ∗s1, const char ∗s2, int n)

  UTF-8 aware strncasecmp - converts to lower case Unicode and tests.
- FL_EXPORT int fl_utf_tolower (const unsigned char ∗str, int len, char ∗buf)

  Converts the string `str` *to its lower case equivalent into buf.*
- FL_EXPORT int fl_utf_toupper (const unsigned char ∗str, int len, char ∗buf)

  Converts the string `str` *to its upper case equivalent into buf.*
- FL_EXPORT int fl_wcwidth (const char ∗src)

  extended wrapper around fl_wcwidth_(unsigned int ucs) function.
- FL_EXPORT int fl_wcwidth_ (unsigned int ucs)

  wrapper to adapt Markus Kuhn's implementation of wcwidth() for FLTK

30.11.1 Detailed Description

fl global Unicode and UTF-8 handling functions declared in <FL/fl_utf8.h>

30.11.2 Macro Definition Documentation

#define ERRORS_TO_CP1252 1

Set to 1 to turn bad UTF-8 bytes in the 0x80-0x9f range into the Unicode index for Microsoft's CP1252 character set. You should also set ERRORS_TO_ISO8859_1. With this a huge amount of more available text (such as all web pages) are correctly converted to Unicode.

#define ERRORS_TO_ISO8859_1 1

Set to 1 to turn bad UTF-8 bytes into ISO-8859-1. If this is zero they are instead turned into the Unicode REPLACEMENT CHARACTER, of value 0xfffd. If this is on fl_utf8decode() will correctly map most (perhaps all) human-readable text that is in ISO-8859-1. This may allow you to completely ignore character sets in your code because virtually everything is either ISO-8859-1 or UTF-8.

#define STRICT_RFC3629 0

A number of Unicode code points are in fact illegal and should not be produced by a UTF-8 converter. Turn this on will replace the bytes in those encodings with errors. If you do this then converting arbitrary 16-bit data to UTF-8 and then back is not an identity, which will probably break a lot of software.

30.11.3 Function Documentation

int fl_access (const char ∗ *f*, int *mode*)

Cross-platform function to test a files access() with a UTF-8 encoded name or value.

This function is especially useful under the MSWindows platform where the standard access() function fails with UTF-8 encoded non-ASCII filenames.

Parameters

`in`	*f*	the UTF-8 encoded filename
`in`	*mode*	the mode to test

Returns

the return value of _waccess() on Windows or access() on other platforms.

int fl_chmod (const char ∗ *f*, int *mode*)

Cross-platform function to set a files mode() with a UTF-8 encoded name or value.

This function is especially useful under the MSWindows platform where the standard chmod() function fails with UTF-8 encoded non-ASCII filenames.

Parameters

`in`	*f*	the UTF-8 encoded filename
`in`	*mode*	the mode to set

Returns

the return value of _wchmod() on Windows or chmod() on other platforms.

FILE ∗ fl_fopen (const char ∗ *f*, const char ∗ *mode*)

Cross-platform function to open files with a UTF-8 encoded name.

This function is especially useful under the MSWindows platform where the standard fopen() function fails with UTF-8 encoded non-ASCII filenames.

Parameters

f	the UTF-8 encoded filename

mode	same as the second argument of the standard fopen() function

Returns

a FILE pointer upon successful completion, or NULL in case of error.

See Also

fl_open().

char ∗ fl_getcwd (char ∗ *b*, int *l*)

Cross-platform function to get the current working directory as a UTF-8 encoded value.

This function is especially useful under the MSWindows platform where the standard _wgetcwd() function returns UTF-16 encoded non-ASCII filenames.

Parameters

b	the buffer to populate
l	the length of the buffer

Returns

the CWD encoded as UTF-8.

char ∗ fl_getenv (const char ∗ *v*)

Cross-platform function to get environment variables with a UTF-8 encoded name or value.

This function is especially useful under the MSWindows platform where non-ASCII environment variables are encoded as wide characters. The returned value of the variable is encoded in UTF-8 as well.

On platforms other than MSWindows this function calls getenv directly. The return value is returned as-is.

Parameters

`in`	*v*	the UTF-8 encoded environment variable

Returns

the environment variable in UTF-8 encoding, or NULL in case of error.

char fl_make_path (const char ∗ *path*)

Cross-platform function to recursively create a path in the file system.

This function creates a `path` in the file system by recursively creating all directories.

void fl_make_path_for_file (const char ∗ *path*)

Cross-platform function to create a path for the file in the file system.

This function strips the filename from the given `path` and creates a path in the file system by recursively creating all directories.

int fl_mkdir (const char ∗ *f*, int *mode*)

Cross-platform function to create a directory with a UTF-8 encoded name.

This function is especially useful on the MSWindows platform where the standard _wmkdir() function expects UTF-16 encoded non-ASCII filenames.

Parameters

in	*f*	the UTF-8 encoded filename
in	*mode*	the mode of the directory

Returns

the return value of _wmkdir() on Windows or mkdir() on other platforms.

unsigned int fl_nonspacing (unsigned int *ucs*)

Returns true if the Unicode character `ucs` is non-spacing.

Non-spacing characters in Unicode are typically combining marks like tilde (∼), diaeresis (¨), or other marks that are added to a base character, for instance 'a' (base character) + '¨' (combining mark) = 'ä' (German Umlaut).

- `http://unicode.org/glossary/#base_character`
- `http://unicode.org/glossary/#nonspacing_mark`
- `http://unicode.org/glossary/#combining_character`

int fl_open (const char * *f*, int *oflags*, ...)

Cross-platform function to open files with a UTF-8 encoded name.

This function is especially useful under the MSWindows platform where the standard open() function fails with UTF-8 encoded non-ASCII filenames.

Parameters

f	the UTF-8 encoded filename
oflags	other arguments are as in the standard open() function

Returns

a file descriptor upon successful completion, or -1 in case of error.

See Also

fl_fopen().

int fl_rename (const char * *f*, const char * *n*)

Cross-platform function to rename a filesystem object using UTF-8 encoded names.

This function is especially useful on the MSWindows platform where the standard _wrename() function expects UTF-16 encoded non-ASCII filenames.

Parameters

in	*f*	the UTF-8 encoded filename to change
in	*n*	the new UTF-8 encoded filename to set

Returns

the return value of _wrename() on Windows or rename() on other platforms.

int fl_rmdir (const char * *f*)

Cross-platform function to remove a directory with a UTF-8 encoded name.

This function is especially useful on the MSWindows platform where the standard _wrmdir() function expects UTF-16 encoded non-ASCII filenames.

Parameters

in	*f*	the UTF-8 encoded filename to remove

Returns

the return value of _wrmdir() on Windows or rmdir() on other platforms.

int fl_stat (const char ∗ *f*, struct stat ∗ *b*)

Cross-platform function to stat() a file using a UTF-8 encoded name or value.

This function is especially useful under the MSWindows platform where the standard stat() function fails with UTF-8 encoded non-ASCII filenames.

Parameters

in	*f*	the UTF-8 encoded filename
	b	the stat struct to populate

Returns

the return value of _wstat() on Windows or stat() on other platforms.

int fl_system (const char ∗ *cmd*)

Cross-platform function to run a system command with a UTF-8 encoded string.

This function is especially useful under the MSWindows platform where non-ASCII program (file) names must be encoded as wide characters.

On platforms other than MSWindows this function calls system() directly.

Parameters

in	*cmd*	the UTF-8 encoded command string

Returns

the return value of _wsystem() on Windows or system() on other platforms.

unsigned fl_ucs_to_Utf16 (const unsigned *ucs*, unsigned short ∗ *dst*, const unsigned *dstlen*)

Convert a single 32-bit Unicode codepoint into an array of 16-bit characters. These are used by some system calls, especially on Windows.

`ucs` is the value to convert.

`dst` points at an array to write, and `dstlen` is the number of locations in this array. At most `dstlen` words will be written, and a 0 terminating word will be added if `dstlen` is large enough. Thus this function will never overwrite the buffer and will attempt return a zero-terminated string if space permits. If `dstlen` is zero then `dst` can be set to NULL and no data is written, but the length is returned.

The return value is the number of 16-bit words that *would* be written to `dst` if it is large enough, not counting any terminating zero.

If the return value is greater than `dstlen` it indicates truncation, you should then allocate a new array of size return+1 and call this again.

Unicode characters in the range 0x10000 to 0x10ffff are converted to ”surrogate pairs” which take two words each (in UTF-16 encoding). Typically, setting `dstlen` to 2 will ensure that any valid Unicode value can be converted, and setting `dstlen` to 3 or more will allow a NULL terminated sequence to be returned.

int fl_unlink (const char ∗ *f*)

Cross-platform function to unlink() (that is, delete) a file using a UTF-8 encoded filename.

This function is especially useful under the MSWindows platform where the standard function expects UTF-16 encoded non-ASCII filenames.

Parameters

f	the filename to unlink

Returns

the return value of _wunlink() on Windows or unlink() on other platforms.

const char ∗ fl_utf8back (const char ∗ *p,* const char ∗ *start,* const char ∗ *end*)

Move p backward until it points to the start of a UTF-8 character. If it already points at the start of one then it is returned unchanged. Any UTF-8 errors are treated as though each byte of the error is an individual character.

start is the start of the string and is used to limit the backwards search for the start of a UTF-8 character.

end is the end of the string and is assumed to be a break between characters. It is assumed to be greater than p.

If you wish to decrement a UTF-8 pointer, pass p-1 to this.

int fl_utf8bytes (unsigned *ucs*)

Return the number of bytes needed to encode the given UCS4 character in UTF-8.

Parameters

in	*ucs*	UCS4 encoded character

Returns

number of bytes required

Returns number of bytes that utf8encode() will use to encode the character ucs.

unsigned fl_utf8decode (const char ∗ *p,* const char ∗ *end,* int ∗ *len*)

Decode a single UTF-8 encoded character starting at *p*. The resulting Unicode value (in the range 0-0x10ffff) is returned, and *len* is set to the number of bytes in the UTF-8 encoding (adding *len* to *p* will point at the next character).

If p points at an illegal UTF-8 encoding, including one that would go past *end*, or where a code uses more bytes than necessary, then ∗(unsigned char∗)p is translated as though it is in the Microsoft CP1252 character set and *len* is set to 1. Treating errors this way allows this to decode almost any ISO-8859-1 or CP1252 text that has been mistakenly placed where UTF-8 is expected, and has proven very useful.

If you want errors to be converted to error characters (as the standards recommend), adding a test to see if the length is unexpectedly 1 will work:

```
if (*p & 0x80) {              // what should be a multibyte encoding
  code = fl_utf8decode(p,end,&len);
  if (len<2) code = 0xFFFD;   // Turn errors into REPLACEMENT CHARACTER
} else {                      // handle the 1-byte UTF-8 encoding:
  code = *p;
  len = 1;
}
```

Direct testing for the 1-byte case (as shown above) will also speed up the scanning of strings where the majority of characters are ASCII.

int fl_utf8encode (unsigned *ucs*, char ∗ *buf*)

Write the UTF-8 encoding of *ucs* into *buf* and return the number of bytes written. Up to 4 bytes may be written. If you know that `ucs` is less than 0x10000 then at most 3 bytes will be written. If you wish to speed this up, remember that anything less than 0x80 is written as a single byte.

If ucs is greater than 0x10ffff this is an illegal character according to RFC 3629. These are converted as though they are 0xFFFD (REPLACEMENT CHARACTER).

RFC 3629 also says many other values for `ucs` are illegal (in the range 0xd800 to 0xdfff, or ending with 0xfffe or 0xffff). However I encode these as though they are legal, so that utf8encode/fl_utf8decode will be the identity for all codes between 0 and 0x10ffff.

unsigned fl_utf8from_mb (char ∗ *dst*, unsigned *dstlen*, const char ∗ *src*, unsigned *srclen*)

Convert a filename from the locale-specific multibyte encoding used by Windows to UTF-8 as used by FLTK.

Up to `dstlen` bytes are written to `dst`, including a null terminator. The return value is the number of bytes that would be written, not counting the null terminator. If greater or equal to `dstlen` then if you malloc a new array of size n+1 you will have the space needed for the entire string. If `dstlen` is zero then nothing is written and this call just measures the storage space needed.

On Unix or on Windows when a UTF-8 locale is in effect, this does not change the data. You may also want to check if fl_utf8test() returns non-zero, so that the filesystem can store filenames in UTF-8 encoding regardless of the locale.

unsigned fl_utf8froma (char ∗ *dst*, unsigned *dstlen*, const char ∗ *src*, unsigned *srclen*)

Convert an ISO-8859-1 (ie normal c-string) byte stream to UTF-8.

It is possible this should convert Microsoft's CP1252 to UTF-8 instead. This would translate the codes in the range 0x80-0x9f to different characters. Currently it does not do this.

Up to `dstlen` bytes are written to `dst`, including a null terminator. The return value is the number of bytes that would be written, not counting the null terminator. If greater or equal to `dstlen` then if you malloc a new array of size n+1 you will have the space needed for the entire string. If `dstlen` is zero then nothing is written and this call just measures the storage space needed.

`srclen` is the number of bytes in `src` to convert.

If the return value equals `srclen` then this indicates that no conversion is necessary, as only ASCII characters are in the string.

unsigned fl_utf8fromwc (char ∗ *dst*, unsigned *dstlen*, const wchar_t ∗ *src*, unsigned *srclen*)

Turn "wide characters" as returned by some system calls (especially on Windows) into UTF-8.

Up to `dstlen` bytes are written to `dst`, including a null terminator. The return value is the number of bytes that would be written, not counting the null terminator. If greater or equal to `dstlen` then if you malloc a new array of size n+1 you will have the space needed for the entire string. If `dstlen` is zero then nothing is written and this call just measures the storage space needed.

`srclen` is the number of words in `src` to convert. On Windows this is not necessarily the number of characters, due to there possibly being "surrogate pairs" in the UTF-16 encoding used. On Unix wchar_t is 32 bits and each location is a character.

On Unix if a `src` word is greater than 0x10ffff then this is an illegal character according to RFC 3629. These are converted as though they are 0xFFFD (REPLACEMENT CHARACTER). Characters in the range 0xd800 to 0xdfff, or ending with 0xfffe or 0xffff are also illegal according to RFC 3629. However I encode these as though they are legal, so that fl_utf8towc will return the original data.

On Windows "surrogate pairs" are converted to a single character and UTF-8 encoded (as 4 bytes). Mismatched halves of surrogate pairs are converted as though they are individual characters.

const char ∗ fl_utf8fwd (const char ∗ *p*, const char ∗ *start*, const char ∗ *end*)

Move `p` forward until it points to the start of a UTF-8 character. If it already points at the start of one then it is returned unchanged. Any UTF-8 errors are treated as though each byte of the error is an individual character.

start is the start of the string and is used to limit the backwards search for the start of a UTF-8 character.

end is the end of the string and is assumed to be a break between characters. It is assumed to be greater than p.

This function is for moving a pointer that was jumped to the middle of a string, such as when doing a binary search for a position. You should use either this or fl_utf8back() depending on which direction your algorithm can handle the pointer moving. Do not use this to scan strings, use fl_utf8decode() instead.

int fl_utf8len (char *c*)

Returns the byte length of the UTF-8 sequence with first byte `c`, or -1 if `c` is not valid.

This function is helpful for finding faulty UTF-8 sequences.

See Also

fl_utf8len1

int fl_utf8len1 (char *c*)

Returns the byte length of the UTF-8 sequence with first byte `c`, or 1 if `c` is not valid.

This function can be used to scan faulty UTF-8 sequences, albeit ignoring invalid codes.

See Also

fl_utf8len

int fl_utf8locale (void)

Return true if the "locale" seems to indicate that UTF-8 encoding is used. If true the fl_utf8to_mb and fl_utf8from_mb don't do anything useful.

It is highly recommended that you change your system so this does return true. On Windows this is done by setting the "codepage" to CP_UTF8. On Unix this is done by setting $LC_CTYPE to a string containing the letters "utf" or "UTF" in it, or by deleting all $LC∗ and $LANG environment variables. In the future it is likely that all non-Asian Unix systems will return true, due to the compatibility of UTF-8 with ISO-8859-1.

int fl_utf8test (const char ∗ *src*, unsigned *srclen*)

Examines the first `srclen` bytes in `src` and returns a verdict on whether it is UTF-8 or not.

- Returns 0 if there is any illegal UTF-8 sequences, using the same rules as fl_utf8decode(). Note that some UCS values considered illegal by RFC 3629, such as 0xffff, are considered legal by this.
- Returns 1 if there are only single-byte characters (ie no bytes have the high bit set). This is legal UTF-8, but also indicates plain ASCII. It also returns 1 if `srclen` is zero.
- Returns 2 if there are only characters less than 0x800.
- Returns 3 if there are only characters less than 0x10000.
- Returns 4 if there are characters in the 0x10000 to 0x10ffff range.

Because there are many illegal sequences in UTF-8, it is almost impossible for a string in another encoding to be confused with UTF-8. This is very useful for transitioning Unix to UTF-8 filenames, you can simply test each filename with this to decide if it is UTF-8 or in the locale encoding. My hope is that if this is done we will be able to cleanly transition to a locale-less encoding.

unsigned fl_utf8to_mb (const char ∗ *src*, unsigned *srclen*, char ∗ *dst*, unsigned *dstlen*)

Convert the UTF-8 used by FLTK to the locale-specific encoding used for filenames (and sometimes used for data in files). Unfortunately due to stupid design you will have to do this as needed for filenames. This is a bug on both Unix and Windows.

Up to `dstlen` bytes are written to `dst`, including a null terminator. The return value is the number of bytes that would be written, not counting the null terminator. If greater or equal to `dstlen` then if you malloc a new array of size n+1 you will have the space needed for the entire string. If `dstlen` is zero then nothing is written and this call just measures the storage space needed.

If fl_utf8locale() returns true then this does not change the data.

unsigned fl_utf8toa (const char ∗ *src*, unsigned *srclen*, char ∗ *dst*, unsigned *dstlen*)

Convert a UTF-8 sequence into an array of 1-byte characters.

If the UTF-8 decodes to a character greater than 0xff then it is replaced with '?'.

Errors in the UTF-8 sequence are converted as individual bytes, same as fl_utf8decode() does. This allows ISO-8859-1 text mistakenly identified as UTF-8 to be printed correctly (and possibly CP1252 on Windows).

`src` points at the UTF-8 sequence, and `srclen` is the number of bytes to convert.

Up to `dstlen` bytes are written to `dst`, including a null terminator. The return value is the number of bytes that would be written, not counting the null terminator. If greater or equal to `dstlen` then if you malloc a new array of size n+1 you will have the space needed for the entire string. If `dstlen` is zero then nothing is written and this call just measures the storage space needed.

unsigned fl_utf8toUtf16 (const char ∗ *src*, unsigned *srclen*, unsigned short ∗ *dst*, unsigned *dstlen*)

Convert a UTF-8 sequence into an array of 16-bit characters. These are used by some system calls, especially on Windows.

`src` points at the UTF-8, and `srclen` is the number of bytes to convert.

`dst` points at an array to write, and `dstlen` is the number of locations in this array. At most `dstlen-1` words will be written there, plus a 0 terminating word. Thus this function will never overwrite the buffer and will always return a zero-terminated string. If `dstlen` is zero then `dst` can be null and no data is written, but the length is returned.

The return value is the number of 16-bit words that *would* be written to `dst` if it were long enough, not counting the terminating zero. If the return value is greater or equal to `dstlen` it indicates truncation, you can then allocate a new array of size return+1 and call this again.

Errors in the UTF-8 are converted as though each byte in the erroneous string is in the Microsoft C-P1252 encoding. This allows ISO-8859-1 text mistakenly identified as UTF-8 to be printed correctly.

Unicode characters in the range 0x10000 to 0x10ffff are converted to "surrogate pairs" which take two words each (this is called UTF-16 encoding).

unsigned fl_utf8towc (const char ∗ *src*, unsigned *srclen*, wchar_t ∗ *dst*, unsigned *dstlen*)

Converts a UTF-8 string into a wide character string.

This function generates 32-bit wchar_t (e.g. "ucs4" as it were) except on Windows where it is equivalent to fl_utf8toUtf16 and returns UTF-16.

`src` points at the UTF-8, and `srclen` is the number of bytes to convert.

`dst` points at an array to write, and `dstlen` is the number of locations in this array. At most `dstlen-1` wchar_t will be written there, plus a 0 terminating wchar_t.

The return value is the number of wchar_t that *would* be written to `dst` if it were long enough, not counting the terminating zero. If the return value is greater or equal to `dstlen` it indicates truncation, you can then allocate a new array of size return+1 and call this again.

Notice that sizeof(wchar_t) is 2 on Windows and is 4 on Linux and most other systems. Where wchar_t is 16 bits, Unicode characters in the range 0x10000 to 0x10ffff are converted to "surrogate pairs" which take two words each (this is called UTF-16 encoding). If wchar_t is 32 bits this rather nasty problem is avoided.

Note that Windows includes Cygwin, i.e. compiled with Cygwin's POSIX layer (cygwin1.dll, –enable-cygwin), either native (GDI) or X11.

int fl_utf_strcasecmp (const char ∗ *s1*, const char ∗ *s2*)

UTF-8 aware strcasecmp - converts to Unicode and tests.

Returns

result of comparison

Return values

0	if the strings are equal
1	if s1 is greater than s2
-1	if s1 is less than s2

int fl_utf_strncasecmp (const char ∗ *s1*, const char ∗ *s2*, int *n*)

UTF-8 aware strncasecmp - converts to lower case Unicode and tests.
Parameters

s1,s2	the UTF-8 strings to compare
n	the maximum number of UTF-8 characters to compare

Returns

result of comparison

Return values

0	if the strings are equal
>0	if s1 is greater than s2
<0	if s1 is less than s2

int fl_utf_tolower (const unsigned char ∗ *str*, int *len*, char ∗ *buf*)

Converts the string `str` to its lower case equivalent into buf.

Warning: to be safe buf length must be at least 3 ∗ len [for 16-bit Unicode]

int fl_utf_toupper (const unsigned char ∗ *str*, int *len*, char ∗ *buf*)

Converts the string `str` to its upper case equivalent into buf.

Warning: to be safe buf length must be at least 3 ∗ len [for 16-bit Unicode]

int fl_wcwidth (const char ∗ *src*)

extended wrapper around fl_wcwidth_(unsigned int ucs) function.
Parameters

in	*src*	pointer to start of UTF-8 byte sequence

Returns

width of character in columns

Depending on build options, this function may map C1 control characters (0x80 to 0x9f) to CP1252, and return the width of that character instead. This is not the same behaviour as fl_wcwidth_(unsigned int ucs) .

Note that other control characters and DEL will still return -1, so if you want different behaviour, you need to test for those characters before calling fl_wcwidth(), and handle them separately.

int fl_wcwidth_ (unsigned int *ucs*)

wrapper to adapt Markus Kuhn's implementation of wcwidth() for FLTK
Parameters

in	*ucs*	Unicode character value

Returns

width of character in columns

See `http://www.cl.cam.ac.uk/~mgk25/ucs/wcwidth.c` for Markus Kuhn's original implementation of wcwidth() and wcswidth() (defined in IEEE Std 1002.1-2001) for Unicode.

WARNING: this function returns widths for "raw" Unicode characters. It does not even try to map C1 control characters (0x80 to 0x9F) to CP1252, and C0/C1 control characters and DEL will return -1. You are advised to use fl_width(const char∗ src) instead.

30.12 Mac OS X-specific symbols

Mac OS X-specific symbols declared in <FL/x.H> or <FL/gl.h>

Classes

- class Fl_Mac_App_Menu

  Mac OS-specific class allowing to customize and localize the application menu.

Functions

- void fl_mac_set_about (Fl_Callback ∗cb, void ∗user_data, int shortcut=0)

  Attaches a callback to the "About myprog" item of the system application menu.
- void fl_open_callback (void(∗cb)(const char ∗))

  Register a function called for each file dropped onto an application icon.
- int gl_texture_pile_height (void)

  Returns the current height of the pile of pre-computed string textures.
- void gl_texture_pile_height (int max)

  Changes the height of the pile of pre-computed string textures.

Variables

- int fl_mac_os_version

  The version number of the running Mac OS X (e.g., 100604 for 10.6.4)
- class Fl_Sys_Menu_Bar ∗ fl_sys_menu_bar

  The system menu bar.

30.12.1 Detailed Description

Mac OS X-specific symbols declared in <FL/x.H> or <FL/gl.h>

See Also

The Apple OS X Interface

30.12.2 Function Documentation

void fl_mac_set_about (Fl_Callback ∗ *cb*, void ∗ *user_data*, int *shortcut* = 0)

Attaches a callback to the "About myprog" item of the system application menu.
Parameters

cb	a callback that will be called by "About myprog" menu item with NULL 1st argument.
user_data	a pointer transmitted as 2nd argument to the callback.
shortcut	optional shortcut to attach to the "About myprog" menu item (e.g., FL_META+'a')

void fl_open_callback (void(∗)(const char ∗) *cb*)

Register a function called for each file dropped onto an application icon.

cb will be called with a single Unix-style file name and path. If multiple files were dropped, *cb* will be called multiple times.

int gl_texture_pile_height (void)

Returns the current height of the pile of pre-computed string textures.

The default value is 100

void gl_texture_pile_height (int *max*)

Changes the height of the pile of pre-computed string textures.

Strings that are often re-displayed can be processed much faster if this pile is set high enough to hold all of them.

Parameters

max	Height of the texture pile

30.13 Common Dialogs classes and functions

Classes

- class Fl_Color_Chooser

  The Fl_Color_Chooser widget provides a standard RGB color chooser.
- class Fl_File_Chooser

  The Fl_File_Chooser widget displays a standard file selection dialog that supports various selection modes.

Functions

- void fl_alert (const char ∗fmt,...)

  Shows an alert message dialog box.
- int fl_ask (const char ∗fmt,...)

  Shows a dialog displaying the `fmt` message, this dialog features 2 yes/no buttons.
- void fl_beep (int type)

  Emits a system beep message.
- int fl_choice (const char ∗fmt, const char ∗b0, const char ∗b1, const char ∗b2,...)

  Shows a dialog displaying the printf style `fmt` message, this dialog features up to 3 customizable choice buttons.
- int fl_color_chooser (const char ∗name, double &r, double &g, double &b, int cmode)

  Pops up a window to let the user pick an arbitrary RGB color.
- int fl_color_chooser (const char ∗name, uchar &r, uchar &g, uchar &b, int cmode)

  Pops up a window to let the user pick an arbitrary RGB color.
- char ∗ fl_dir_chooser (const char ∗message, const char ∗fname, int relative)

  Shows a file chooser dialog and gets a directory.
- char ∗ fl_file_chooser (const char ∗message, const char ∗pat, const char ∗fname, int relative)

  Shows a file chooser dialog and gets a filename.
- void fl_file_chooser_callback (void(∗cb)(const char ∗))

  Set the file chooser callback.
- void fl_file_chooser_ok_label (const char ∗l)

  Set the "OK" button label.
- const char ∗ fl_input (const char ∗fmt, const char ∗defstr,...)

  Shows an input dialog displaying the `fmt` message.
- void fl_message (const char ∗fmt,...)

  Shows an information message dialog box.
- void fl_message_hotspot (int enable)

  Sets whether or not to move the common message box used in many common dialogs like fl_message(), fl_alert(), fl_ask(), fl_choice(), fl_input(), fl_password() to follow the mouse pointer.
- int fl_message_hotspot (void)

  Gets whether or not to move the common message box used in many common dialogs like fl_message(), fl_alert(), fl_ask(), fl_choice(), fl_input(), fl_password() to follow the mouse pointer.
- Fl_Widget ∗ fl_message_icon ()

  Gets the Fl_Box icon container of the current default dialog used in many common dialogs like fl_message(), fl_alert(), fl_ask(), fl_choice(), fl_input(), fl_password()
- void fl_message_title (const char ∗title)

  Sets the title of the dialog window used in many common dialogs.
- void fl_message_title_default (const char ∗title)

  Sets the default title of the dialog window used in many common dialogs.
- const char ∗ fl_password (const char ∗fmt, const char ∗defstr,...)

  Shows an input dialog displaying the `fmt` message.

Variables

- static void(∗ Fl::error)(const char ∗,...) = ::error

  FLTK calls Fl::error() to output a normal error message.
- static void(∗ Fl::fatal)(const char ∗,...) = ::fatal

  FLTK calls Fl::fatal() to output a fatal error message.
- const char ∗ fl_cancel = ”Cancel”

  string pointer used in common dialogs, you can change it to another language
- const char ∗ fl_close = ”Close”

  string pointer used in common dialogs, you can change it to another language
- const char ∗ fl_no = ”No”

  string pointer used in common dialogs, you can change it to another language
- const char ∗ fl_ok = ”OK”

  string pointer used in common dialogs, you can change it to another language
- const char ∗ fl_yes = ”Yes”

  string pointer used in common dialogs, you can change it to another language
- static void(∗ Fl::warning)(const char ∗,...) = ::warning

  FLTK calls Fl::warning() to output a warning message.

30.13.1 Detailed Description

30.13.2 Function Documentation

void fl_alert (const char ∗ *fmt,* ...)

Shows an alert message dialog box.

Note

Common dialog boxes are application modal. No more than one common dialog box can be open at any time. Requests for additional dialog boxes are ignored.
#include <FL/fl_ask.H>

Parameters

in	*fmt*	can be used as an sprintf-like format and variables for the message text

int fl_ask (const char ∗ *fmt,* ...)

Shows a dialog displaying the `fmt` message, this dialog features 2 yes/no buttons.

Note

Common dialog boxes are application modal. No more than one common dialog box can be open at any time. Requests for additional dialog boxes are ignored.
#include <FL/fl_ask.H>

Parameters

in	*fmt*	can be used as an sprintf-like format and variables for the message text

Return values

0	if the no button is selected or another dialog box is still open
1	if yes is selected

Deprecated fl_ask() is deprecated since it uses "Yes" and "No" for the buttons which does not conform to the current FLTK Human Interface Guidelines. Use fl_choice() with the appropriate verbs instead.

void fl_beep (int *type*)

Emits a system beep message.

Parameters

`in`	*type*	The beep type from the Fl_Beep enumeration.

Note

#include <FL/fl_ask.H>

int fl_choice (const char ∗ *fmt,* const char ∗ *b0,* const char ∗ *b1,* const char ∗ *b2,* ...)

Shows a dialog displaying the printf style `fmt` message, this dialog features up to 3 customizable choice buttons.

Note

Common dialog boxes are application modal. No more than one common dialog box can be open at any time. Requests for additional dialog boxes are ignored.
#include <FL/fl_ask.H>

Three choices with printf() style formatting:

```
int num_msgs = GetNumberOfMessages();
switch ( fl_choice("What to do with %d messages?", "Send", "Save", "Delete", num_msgs) ) {
  case 0: .. // Send
  case 1: .. // Save (default)
  case 2: .. // Delete
  ..
}
```

Three choice example:

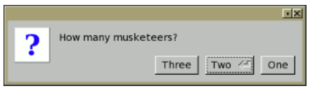


Figure 30.2: fl_choice() three choices

```
switch ( fl_choice("How many musketeers?", "One", "Two", "Three") ) {
  case 0: .. // One
  case 1: .. // Two (default)
  case 2: .. // Three
}
```

Two choice example:

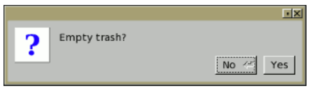
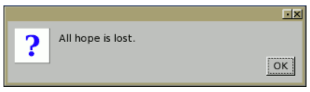


Figure 30.3: fl_choice() two choices

```
switch ( fl_choice("Empty trash?", "Yes", "No", 0) ) {
  case 0: .. // Yes
  case 1: .. // No (default)
}
```

One choice example:



All hope is lost.

Figure 30.4: fl_choice() one choice

```
fl_choice("All hope is lost.", "OK", 0, 0);   // "OK" default
```

Parameters

`in`	*fmt*	can be used as an sprintf-like format and variables for the message text
`in`	*b0*	text label of button 0
`in`	*b1*	text label of button 1 (can be 0)
`in`	*b2*	text label of button 2 (can be 0)

Return values

0	if the first button with `b0` text is pushed or another dialog box is still open
1	if the second button with `b1` text is pushed
2	if the third button with `b2` text is pushed

int fl_color_chooser (const char ∗ ***name,* double & *r,* double & *g,* double & *b,* int *cmode*)**
`[related]`

Pops up a window to let the user pick an arbitrary RGB color.

Note

#include <FL/Fl_Color_Chooser.H>

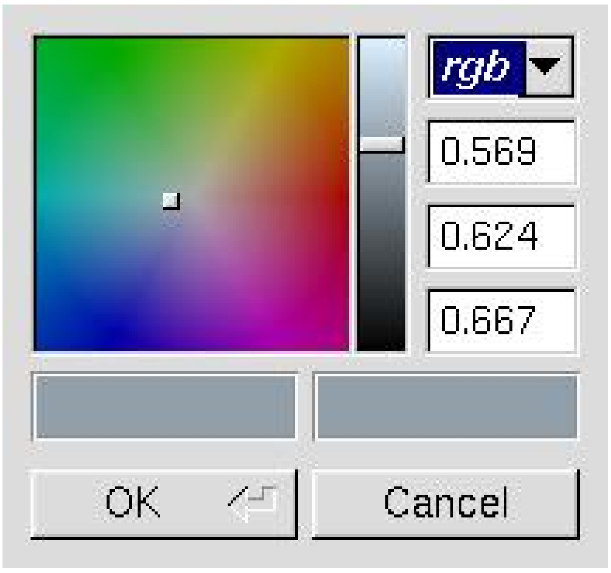


Figure 30.5: fl_color_chooser

Parameters

`in`	*name*	Title label for the window
`in,out`	*r,g,b*	Color components in the range 0.0 to 1.0.
`in`	*cmode*	Optional mode for color chooser. See mode(int). Default -1 if none (rgb mode).

Return values

1	if user confirms the selection
0	if user cancels the dialog

int fl_color_chooser (const char ∗ *name*, uchar & *r*, uchar & *g*, uchar & *b*, int *cmode*) `[related]`

Pops up a window to let the user pick an arbitrary RGB color.

Note

#include <FL/Fl_Color_Chooser.H>

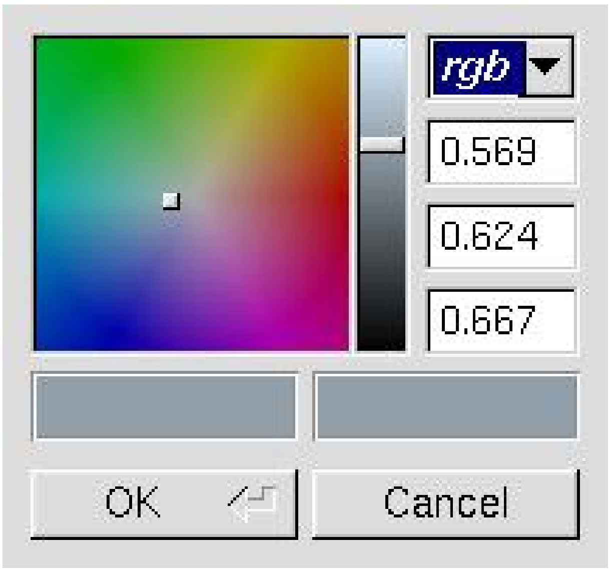


Figure 30.6: fl_color_chooser

Parameters

`in`	*name*	Title label for the window
`in,out`	*r,g,b*	Color components in the range 0 to 255.
`in`	*cmode*	Optional mode for color chooser. See mode(int). Default -1 if none (rgb mode).

Return values

1	if user confirms the selection
0	if user cancels the dialog

char ∗ fl_dir_chooser (const char ∗ *message*, const char ∗ *fname*, int *relative*) `[related]`

Shows a file chooser dialog and gets a directory.

Note

#include <FL/Fl_File_Chooser.H>

Parameters

in	*message*	title bar text
in	*fname*	initial/default directory name
in	*relative*	0 for absolute path return, relative otherwise

Returns

the directory path string chosen by the user or NULL if user cancels

char ∗ fl_file_chooser (const char ∗ *message*, const char ∗ *pat*, const char ∗ *fname*, int *relative*) `[related]`

Shows a file chooser dialog and gets a filename.

Note

#include <FL/Fl_File_Chooser.H>

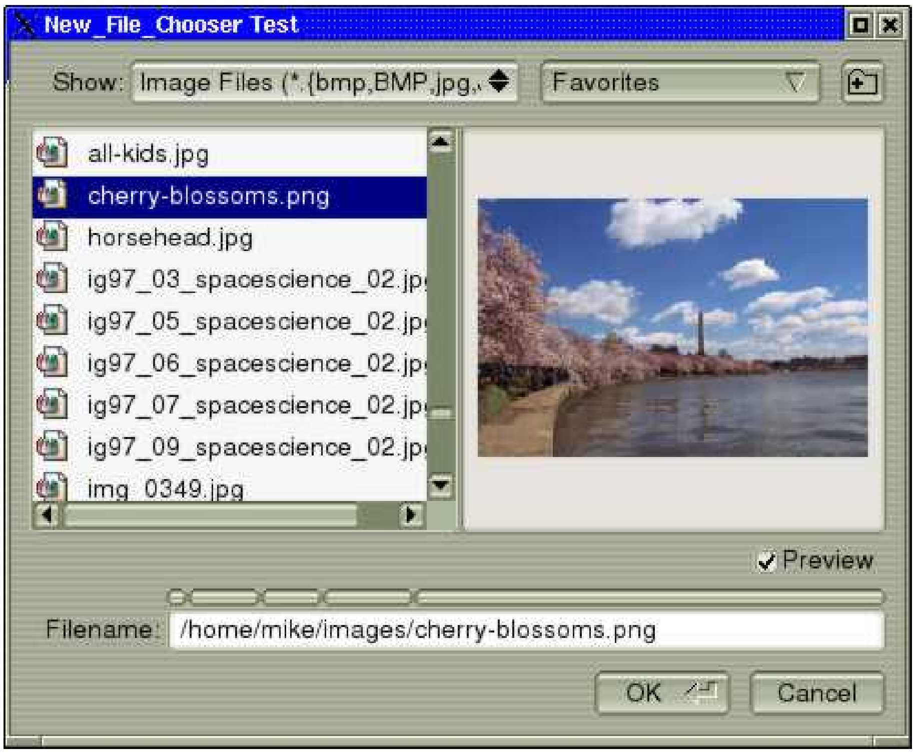

New_File_Chooser Test
Show: Image Files (*.{bmp,BMP,jpg,
Favorites
all-kids.jpg
cherry-blossoms.png
horsehead.jpg
ig97_03_spacescience_02.jp
ig97_05_spacescience_02.jp
ig97_06_spacescience_02.jp
ig97_07_spacescience_02.jp
ig97_09_spacescience_02.jp
img_0349.jpg
Preview
Filename: /home/mike/images/cherry-blossoms.png
OK
Cancel

Figure 30.7: Fl_File_Chooser

Parameters

`in`	*message*	text in title bar
`in`	*pat*	filename pattern filter
`in`	*fname*	initial/default filename selection
`in`	*relative*	0 for absolute path name, relative path name otherwise

Returns

the user selected filename, in absolute or relative format or NULL if user cancels

void fl_file_chooser_callback (void(∗)(const char ∗) ***cb*** **)** `[related]`

Set the file chooser callback.

Note

#include <FL/Fl_File_Chooser.H>

void fl_file_chooser_ok_label (const char ∗ ***l*** **)** `[related]`

Set the "OK" button label.

Note

#include <FL/Fl_File_Chooser.H>

const char∗ fl_input (const char ∗ ***fmt,*** **const char ∗** ***defstr,*** **...)**

Shows an input dialog displaying the `fmt` message.

Note

Common dialog boxes are application modal. No more than one common dialog box can be open at any time. Requests for additional dialog boxes are ignored.
#include <FL/fl_ask.H>

Parameters

`in`	*fmt*	can be used as an sprintf-like format and variables for the message text
`in`	*defstr*	defines the default returned string if no text is entered

Returns

the user string input if OK was pushed, NULL if Cancel was pushed or another dialog box was still open

void fl_message (const char ∗ ***fmt,*** **...)**

Shows an information message dialog box.

Note

Common dialog boxes are application modal. No more than one common dialog box can be open at any time. Requests for additional dialog boxes are ignored.
#include <FL/fl_ask.H>

Parameters

`in`	*fmt*	can be used as an sprintf-like format and variables for the message text

void fl_message_hotspot (int *enable*)

Sets whether or not to move the common message box used in many common dialogs like fl_message(), fl_alert(), fl_ask(), fl_choice(), fl_input(), fl_password() to follow the mouse pointer.

The default is *enabled*, so that the default button is the hotspot and appears at the mouse position.

Note

#include <FL/fl_ask.H>

Parameters

`in`	*enable*	non-zero enables hotspot behavior, 0 disables hotspot

int fl_message_hotspot (void)

Gets whether or not to move the common message box used in many common dialogs like fl_message(), fl_alert(), fl_ask(), fl_choice(), fl_input(), fl_password() to follow the mouse pointer.

Note

#include <FL/fl_ask.H>

Returns

0 if disable, non-zero otherwise

See Also

fl_message_hotspot(int)

Fl_Widget∗ fl_message_icon ( )

Gets the Fl_Box icon container of the current default dialog used in many common dialogs like fl_message(), fl_alert(), fl_ask(), fl_choice(), fl_input(), fl_password()

Note

#include <FL/fl_ask.H>

void fl_message_title (const char ∗ *title*)

Sets the title of the dialog window used in many common dialogs.

This window `title` will be used in the next call of one of the common dialogs like fl_message(), fl_alert(), fl_ask(), fl_choice(), fl_input(), fl_password().

The `title` string is copied internally, so that you can use a local variable or free the string immediately after this call. It applies only to the **next** call of one of the common dialogs and will be reset to an empty title (the default for all dialogs) after that call.

Note

#include <FL/fl_ask.H>

Parameters

`in`	*title*	window label, string copied internally

void fl_message_title_default (const char ∗ *title*)

Sets the default title of the dialog window used in many common dialogs.

This window `title` will be used in all subsequent calls of one of the common dialogs like fl_message(), fl_alert(), fl_ask(), fl_choice(), fl_input(), fl_password(), unless a specific title has been set with fl_message_title(const char ∗title).

The default is no title. You can override the default title for a single dialog with fl_message_title(const char ∗title).

The `title` string is copied internally, so that you can use a local variable or free the string immediately after this call.

Note

#include <FL/fl_ask.H>

Parameters

`in`	*title*	default window label, string copied internally

const char∗ fl_password (const char ∗ *fmt,* const char ∗ *defstr,* ...)

Shows an input dialog displaying the `fmt` message.

Like fl_input() except the input text is not shown, '∗' characters are displayed instead.

Note

Common dialog boxes are application modal. No more than one common dialog box can be open at any time. Requests for additional dialog boxes are ignored.
#include <FL/fl_ask.H>

Parameters

`in`	*fmt*	can be used as an sprintf-like format and variables for the message text
`in`	*defstr*	defines the default returned string if no text is entered

Returns

the user string input if OK was pushed, NULL if Cancel was pushed or aother dialog box was still open

30.13.3 Variable Documentation

void(∗ Fl::error)(const char ∗format,...) = ::error `[static]`

FLTK calls Fl::error() to output a normal error message.

The default version on Windows displays the error message in a MessageBox window.

The default version on all other platforms prints the error message to stderr.

You can override the behavior by setting the function pointer to your own routine.

Fl::error() means there is a recoverable error such as the inability to read an image file. The default implementation returns after displaying the message.

Note

#include <FL/Fl.H>

void(∗ Fl::fatal)(const char ∗format,...) = ::fatal `[static]`

FLTK calls Fl::fatal() to output a fatal error message.

The default version on Windows displays the error message in a MessageBox window.

The default version on all other platforms prints the error message to stderr.

You can override the behavior by setting the function pointer to your own routine.

Fl::fatal() must not return, as FLTK is in an unusable state, however your version may be able to use longjmp or an exception to continue, as long as it does not call FLTK again. The default implementation exits with status 1 after displaying the message.

Note

#include <FL/Fl.H>

void(∗ Fl::warning)(const char ∗format,...) = ::warning `[static]`

FLTK calls Fl::warning() to output a warning message.

The default version on Windows returns *without* printing a warning message, because Windows programs normally don't have stderr (a console window) enabled.

The default version on all other platforms prints the warning message to stderr.

You can override the behavior by setting the function pointer to your own routine.

Fl::warning() means that there was a recoverable problem, the display may be messed up, but the user can probably keep working - all X protocol errors call this, for example. The default implementation returns after displaying the message.

Note

#include <FL/Fl.H>

30.14 File names and URI utility functions

File names and URI functions defined in <FL/filename.H>

Macros

- #define **fl_dirent_h_cyclic_include**
- #define FL_PATH_MAX 2048

    all path buffers should use this length

Typedefs

- typedef int(Fl_File_Sort_F)(struct dirent ∗∗, struct dirent ∗∗)

    File sorting function.

Functions

- FL_EXPORT void fl_decode_uri (char ∗uri)

    Decodes a URL-encoded string.
- FL_EXPORT int fl_filename_absolute (char ∗to, int tolen, const char ∗from)

    Makes a filename absolute from a relative filename.
- FL_EXPORT int fl_filename_expand (char ∗to, int tolen, const char ∗from)

    Expands a filename containing shell variables and tilde (∼).
- FL_EXPORT const char ∗ fl_filename_ext (const char ∗buf)

    Gets the extensions of a filename.
- FL_EXPORT void fl_filename_free_list (struct dirent ∗∗∗l, int n)

    Free the list of filenames that is generated by fl_filename_list().
- FL_EXPORT int fl_filename_isdir (const char ∗name)

    Determines if a file exists and is a directory from its filename.
- FL_EXPORT int fl_filename_list (const char ∗d, struct dirent ∗∗∗l, Fl_File_Sort_F ∗s=fl_numericsort)

    Portable and const-correct wrapper for the scandir() function.
- FL_EXPORT int fl_filename_match (const char ∗name, const char ∗pattern)

    Checks if a string `s` matches a pattern `p`.
- FL_EXPORT const char ∗ fl_filename_name (const char ∗filename)

    Gets the file name from a path.
- FL_EXPORT int fl_filename_relative (char ∗to, int tolen, const char ∗from)

    Makes a filename relative to the current working directory.
- FL_EXPORT char ∗ fl_filename_setext (char ∗to, int tolen, const char ∗ext)

    Replaces the extension in `buf` of max.
- FL_EXPORT int fl_open_uri (const char ∗uri, char ∗msg, int msglen)

    Opens the specified Uniform Resource Identifier (URI).

30.14.1 Detailed Description

File names and URI functions defined in <FL/filename.H>

30.14.2 Typedef Documentation

typedef int(Fl_File_Sort_F)(struct dirent ∗∗, struct dirent ∗∗)

File sorting function.

See Also

fl_filename_list()

30.14.3 Function Documentation

void fl_decode_uri (char ∗ *uri*)

Decodes a URL-encoded string.

In a Uniform Resource Identifier (URI), all non-ASCII bytes and several others (e.g., '<', ", ' ') are URL-encoded using 3 bytes by "%XY" where XY is the hexadecimal value of the byte. This function decodes the URI restoring its original UTF-8 encoded content. Decoding is done in-place.

FL_EXPORT int fl_filename_absolute (char ∗ *to,* int *tolen,* const char ∗ *from*)

Makes a filename absolute from a relative filename.

```
#include <FL/filename.H>
[..]
chdir("/var/tmp");
fl_filename_absolute(out, sizeof(out), "foo.txt");         // out="/var/tmp/foo.txt"
fl_filename_absolute(out, sizeof(out), "./foo.txt");       // out="/var/tmp/foo.txt"
fl_filename_absolute(out, sizeof(out), "../log/messages"); // out="/var/log/messages"
```

Parameters

out	*to*	resulting absolute filename
in	*tolen*	size of the absolute filename buffer
in	*from*	relative filename

Returns

0 if no change, non zero otherwise

FL_EXPORT int fl_filename_expand (char ∗ *to,* int *tolen,* const char ∗ *from*)

Expands a filename containing shell variables and tilde (∼).

Currently handles these variants:

```
"~username"               // if 'username' does not exist, result will be unchanged
"~/file"
"$VARNAME"                // does NOT handle ${VARNAME}
```

Examples:

```
#include <FL/filename.H>
[..]
putenv("TMPDIR=/var/tmp");
fl_filename_expand(out, sizeof(out), "~fred/.cshrc");     // out="/usr/fred/.cshrc"
fl_filename_expand(out, sizeof(out), "~/.cshrc");         // out="/usr/<yourname>/.cshrc"
fl_filename_expand(out, sizeof(out), "$TMPDIR/foo.txt");  // out="/var/tmp/foo.txt"
```

Parameters

out	*to*	resulting expanded filename
in	*tolen*	size of the expanded filename buffer
in	*from*	filename containing shell variables

Returns

0 if no change, non zero otherwise

FL_EXPORT const char∗ fl_filename_ext (const char ∗ *buf*)

Gets the extensions of a filename.

```
#include <FL/filename.H>
[..]
const char *out;
out = fl_filename_ext("/some/path/foo.txt");        // result: ".txt"
out = fl_filename_ext("/some/path/foo");            // result: NULL
```

Parameters

in	*buf*	the filename to be parsed

Returns

a pointer to the extension (including '.') if any or NULL otherwise

FL_EXPORT void fl_filename_free_list (struct dirent ∗∗∗ *list*, int *n*)

Free the list of filenames that is generated by fl_filename_list().

Free everything that was allocated by a previous call to fl_filename_list(). Use the return values as parameters for this function.

Parameters

in,out	*list*	table containing the resulting directory listing
in	*n*	number of entries in the list

FL_EXPORT int fl_filename_isdir (const char ∗ *n*)

Determines if a file exists and is a directory from its filename.

```
#include <FL/filename.H>
[..]
fl_filename_isdir("/etc");              // returns non-zero
fl_filename_isdir("/etc/hosts");        // returns 0
```

Parameters

in	*n*	the filename to parse

Returns

non zero if file exists and is a directory, zero otherwise

FL_EXPORT int fl_filename_list (const char ∗ *d*, dirent ∗∗∗ *list*, Fl_File_Sort_F ∗ *sort*)

Portable and const-correct wrapper for the scandir() function.

For each file in that directory a ”dirent” structure is created. The only portable thing about a dirent is that dirent.d_name is the nul-terminated file name. An pointers array to these dirent's is created and a pointer to the array is returned in ∗list. The number of entries is given as a return value. If there is an error reading the directory a number less than zero is returned, and errno has the reason; errno does not work under WIN32.

Include:

```
#include <FL/filename.H>
```

Parameters

in	*d*	the name of the directory to list. It does not matter if it has a trailing slash.
out	*list*	table containing the resulting directory listing
in	*sort*	sorting functor: • fl_alphasort: The files are sorted in ascending alphabetical order; upper and lowercase letters are compared according to their ASCII ordering uppercase before lowercase. • fl_casealphasort: The files are sorted in ascending alphabetical order; upper and lowercase letters are compared equally case is not significant. • fl_casenumericsort: The files are sorted in ascending ”alphanumeric” order, where an attempt is made to put unpadded numbers in consecutive order; upper and lowercase letters are compared equally case is not significant. • fl_numericsort: The files are sorted in ascending ”alphanumeric” order, where an attempt is made to put unpadded numbers in consecutive order; upper and lowercase letters are compared according to their ASCII ordering - uppercase before lowercase.

Returns

the number of entries if no error, a negative value otherwise.

FL_EXPORT int fl_filename_match (const char ∗ *s*, const char ∗ *p*)

Checks if a string `s` matches a pattern `p`.

The following syntax is used for the pattern:

- ∗ matches any sequence of 0 or more characters.
- ? matches any single character.
- [set] matches any character in the set. Set can contain any single characters, or a-z to represent a range. To match] or - they must be the first characters. To match ^ or ! they must not be the first characters.
- [^set] or [!set] matches any character not in the set.
- {X|Y|Z} or {X,Y,Z} matches any one of the subexpressions literally.

- \x quotes the character x so it has no special meaning.
- x all other characters must be matched exactly.

Include:

```
#include <FL/filename.H>
```

Parameters

in	*s*	the string to check for a match
in	*p*	the string pattern

Returns

non zero if the string matches the pattern

FL_EXPORT const char∗ fl_filename_name (const char ∗ *filename*)

Gets the file name from a path.
Similar to basename(3), exceptions shown below.

```
#include <FL/filename.H>
[..]
const char *out;
out = fl_filename_name("/usr/lib");     // out="lib"
out = fl_filename_name("/usr/");        // out=""      (basename(3) returns "usr" instead)
out = fl_filename_name("/usr");         // out="usr"
out = fl_filename_name("/");            // out=""      (basename(3) returns "/" instead)
out = fl_filename_name(".");            // out="."
out = fl_filename_name("..");           // out=".."
```

Returns

a pointer to the char after the last slash, or to `filename` if there is none.

FL_EXPORT int fl_filename_relative (char ∗ *to,* int *tolen,* const char ∗ *from*)

Makes a filename relative to the current working directory.

```
#include <FL/filename.H>
[..]
chdir("/var/tmp/somedir");       // set cwd to /var/tmp/somedir
[..]
char out[FL_PATH_MAX];
fl_filename_relative(out, sizeof(out), "/var/tmp/somedir/foo.txt");  // out="foo.txt",
        return=1
fl_filename_relative(out, sizeof(out), "/var/tmp/foo.txt");          //
       out="../foo.txt", return=1
fl_filename_relative(out, sizeof(out), "foo.txt");                   // out="foo.txt",
        return=0 (no change)
fl_filename_relative(out, sizeof(out), "./foo.txt");                 //
       out="./foo.txt",  return=0 (no change)
fl_filename_relative(out, sizeof(out), "../foo.txt");                //
       out="../foo.txt", return=0 (no change)
```

Parameters

out	*to*	resulting relative filename
in	*tolen*	size of the relative filename buffer
in	*from*	absolute filename

Returns

0 if no change, non zero otherwise

FL_EXPORT char∗ fl_filename_setext (char ∗ *buf,* int *buflen,* const char ∗ *ext*)

Replaces the extension in `buf` of max.
size `buflen` with the extension in `ext`.
If there's no '.' in `buf`, `ext` is appended.
If `ext` is NULL, behaves as if it were an empty string ("").
Example

```
#include <FL/filename.H>
[..]
char buf[FL_PATH_MAX] = "/path/myfile.cxx";
fl_filename_setext(buf, sizeof(buf), ".txt");      // buf[] becomes "/path/myfile.txt"
```

Returns

buf itself for calling convenience.

int fl_open_uri (const char ∗ *uri,* char ∗ *msg,* int *msglen*)

Opens the specified Uniform Resource Identifier (URI).
Uses an operating-system dependent program or interface. For URIs using the "ftp", "http", or "https" schemes, the system default web browser is used to open the URI, while "mailto" and "news" URIs are typically opened using the system default mail reader and "file" URIs are opened using the file system navigator.
On success, the (optional) msg buffer is filled with the command that was run to open the URI; on Windows, this will always be "open uri".
On failure, the msg buffer is filled with an English error message.

Note

Platform Specific Issues: Windows
With "file:" based URIs on Windows, you may encounter issues with anchors being ignored. Example-: "file:///c:/some/index.html#anchor" may open in the browser without the "#anchor" suffix. The behavior seems to vary across different Windows versions. Workaround: open a link to a separate html file that redirects to the desired "file:" URI.

Example

```
#include <FL/filename.H>
[..]
char errmsg[512];
if ( !fl_open_uri("http://google.com/", errmsg, sizeof(errmsg)) ) {
    char warnmsg[768];
    sprintf(warnmsg, "Error: %s", errmsg);
    fl_alert(warnmsg);
}
```

Parameters

uri	The URI to open
msg	Optional buffer which contains the command or error message
msglen	Length of optional buffer

Returns

1 on success, 0 on failure

Chapter 31

Class Documentation

31.1 Fl_Preferences::Entry Struct Reference

Public Attributes

- char ∗ **name**
- char ∗ **value**

The documentation for this struct was generated from the following file:

- Fl_Preferences.H

31.2 Fl Class Reference

The Fl is the FLTK global (static) class containing state information and global methods for the current application.

```
#include <Fl.H>
```

Public Types

- enum Fl_Option {
  OPTION_ARROW_FOCUS = 0, OPTION_VISIBLE_FOCUS, OPTION_DND_TEXT, OPTION_S-HOW_TOOLTIPS,
  OPTION_FNFC_USES_GTK, OPTION_LAST }

  Enumerator for global FLTK options.

Static Public Member Functions

- static int abi_check (const int val=FL_ABI_VERSION)

  Returns whether the runtime library ABI version is correct.
- static int abi_version ()

  Returns the compiled-in value of the FL_ABI_VERSION constant.
- static int add_awake_handler_ (Fl_Awake_Handler, void ∗)

  Adds an awake handler for use in awake().
- static void add_check (Fl_Timeout_Handler, void ∗=0)

  FLTK will call this callback just before it flushes the display and waits for events.
- static void add_clipboard_notify (Fl_Clipboard_Notify_Handler h, void ∗data=0)

  FLTK will call the registered callback whenever there is a change to the selection buffer or the clipboard.

- static void add_fd (int fd, int when, Fl_FD_Handler cb, void ∗=0)

  Adds file descriptor fd to listen to.
- static void add_fd (int fd, Fl_FD_Handler cb, void ∗=0)

  See void add_fd(int fd, int when, Fl_FD_Handler cb, void∗ = 0)
- static void add_handler (Fl_Event_Handler h)

  Install a function to parse unrecognized events.
- static void add_idle (Fl_Idle_Handler cb, void ∗data=0)

  Adds a callback function that is called every time by Fl::wait() and also makes it act as though the timeout is zero (this makes Fl::wait() return immediately, so if it is in a loop it is called repeatedly, and thus the idle fucntion is called repeatedly).
- static void add_system_handler (Fl_System_Handler h, void ∗data)

  Install a function to intercept system events.
- static void add_timeout (double t, Fl_Timeout_Handler, void ∗=0)

  Adds a one-shot timeout callback.
- static int api_version ()

  Returns the compiled-in value of the FL_API_VERSION constant.
- static int arg (int argc, char ∗∗argv, int &i)

  Parse a single switch from `argv`, starting at word `i`.
- static int args (int argc, char ∗∗argv, int &i, Fl_Args_Handler cb=0)

  Parse command line switches using the `cb` argument handler.
- static void args (int argc, char ∗∗argv)

  Parse all command line switches matching standard FLTK options only.
- static void awake (void ∗message=0)

  Sends a message pointer to the main thread, causing any pending Fl::wait() call to terminate so that the main thread can retrieve the message and any pending redraws can be processed.
- static int awake (Fl_Awake_Handler cb, void ∗message=0)

  See void awake(void∗ message=0).
- static void background (uchar, uchar, uchar)

  Changes fl_color(FL_BACKGROUND_COLOR) to the given color, and changes the gray ramp from 32 to 56 to black to white.
- static void background2 (uchar, uchar, uchar)

  Changes the alternative background color.
- static Fl_Widget ∗ belowmouse ()

  Gets the widget that is below the mouse.
- static void belowmouse (Fl_Widget ∗)

  Sets the widget that is below the mouse.
- static Fl_Color box_color (Fl_Color)

  Gets the drawing color to be used for the background of a box.
- static int box_dh (Fl_Boxtype)

  Returns the height offset for the given boxtype.
- static int box_dw (Fl_Boxtype)

  Returns the width offset for the given boxtype.
- static int box_dx (Fl_Boxtype)

  Returns the X offset for the given boxtype.
- static int box_dy (Fl_Boxtype)

  Returns the Y offset for the given boxtype.
- static void cairo_autolink_context (bool alink)

when FLTK_HAVE_CAIRO is defined and cairo_autolink_context() is true, any current window dc is linked to a current cairo context.

- static bool cairo_autolink_context ()

  Gets the current autolink mode for cairo support.
- static cairo_t ∗ cairo_cc ()

  Gets the current cairo context linked with a fltk window.
- static void cairo_cc (cairo_t ∗c, bool own=false)

  Sets the current cairo context to `c`*.*
- static cairo_t ∗ cairo_make_current (Fl_Window ∗w)

  Provides a corresponding cairo context for window wi.
- static int check ()

  Same as Fl::wait(0).
- static void clear_widget_pointer (Fl_Widget const ∗w)

  Clears a widget pointer in the watch list.
- static int clipboard_contains (const char ∗type)

  Returns non 0 if the clipboard contains data matching `type`*.*
- static int compose (int &del)

  Any text editing widget should call this for each FL_KEYBOARD event.
- static void compose_reset ()

  If the user moves the cursor, be sure to call Fl::compose_reset().
- static void copy (const char ∗stuff, int len, int destination=0, const char ∗type=Fl::clipboard_plain_-text)

  Copies the data pointed to by `stuff` *to the selection buffer (*`destination` *is 0), the clipboard (*`destination` *is 1), or both (*`destination` *is 2).*
- static void damage (int d)

  If true then flush() will do something.
- static int damage ()

  If true then flush() will do something.
- static void default_atclose (Fl_Window ∗, void ∗)

  Default callback for window widgets.
- static void delete_widget (Fl_Widget ∗w)

  Schedules a widget for deletion at the next call to the event loop.
- static void disable_im ()

  Disables the system input methods facilities.
- static void display (const char ∗)

  Sets the X display to use for all windows.
- static int dnd ()

  Initiate a Drag And Drop operation.
- static void dnd_text_ops (int v)

  Gets or sets whether drag and drop text operations are supported.
- static int dnd_text_ops ()

  Gets or sets whether drag and drop text operations are supported.
- static void do_widget_deletion ()

  Deletes widgets previously scheduled for deletion.
- static int draw_box_active ()

  Determines if the currently drawn box is active or inactive.
- static void enable_im ()

Enables the system input methods facilities.

- static int event ()

  Returns the last event that was processed.
- static int event_alt ()

  Returns non-zero if the Alt key is pressed.
- static int event_button ()

  Gets which particular mouse button caused the current event.
- static int event_button1 ()

  Returns non-zero if mouse button 1 is currently held down.
- static int event_button2 ()

  Returns non-zero if button 2 is currently held down.
- static int event_button3 ()

  Returns non-zero if button 3 is currently held down.
- static int event_buttons ()

  Returns the mouse buttons state bits; if non-zero, then at least one button is pressed now.
- static int event_clicks ()

  Returns non zero if we had a double click event.
- static void event_clicks (int i)

  Manually sets the number returned by Fl::event_clicks().
- static void ∗ event_clipboard ()

  During an FL_PASTE event of non-textual data, returns a pointer to the pasted data.
- static const char ∗ event_clipboard_type ()

  Returns the type of the pasted data during an FL_PASTE event.
- static int event_command ()

  Returns non-zero if the FL_COMMAND key is pressed, either FL_CTRL or on OSX FL_META.
- static int event_ctrl ()

  Returns non-zero if the Control key is pressed.
- static void event_dispatch (Fl_Event_Dispatch d)

  Set a new event dispatch function.
- static Fl_Event_Dispatch event_dispatch ()

  Return the current event dispatch function.
- static int event_dx ()

  Returns the current horizontal mouse scrolling associated with the FL_MOUSEWHEEL event.
- static int event_dy ()

  Returns the current vertical mouse scrolling associated with the FL_MOUSEWHEEL event.
- static int event_inside (int, int, int, int)

  Returns whether or not the mouse event is inside the given rectangle.
- static int event_inside (const Fl_Widget ∗)

  Returns whether or not the mouse event is inside a given child widget.
- static int event_is_click ()

  Returns non-zero if the mouse has not moved far enough and not enough time has passed since the last FL_PUSH or FL_KEYBOARD event for it to be considered a "drag" rather than a "click".
- static void event_is_click (int i)

  Clears the value returned by Fl::event_is_click().
- static int event_key ()

  Gets which key on the keyboard was last pushed.
- static int event_key (int key)

Returns true if the given `key` was held down (or pressed) during the last event.

- static int event_length ()

  Returns the length of the text in Fl::event_text().
- static int event_original_key ()

  Returns the keycode of the last key event, regardless of the NumLock state.
- static int event_shift ()

  Returns non-zero if the Shift key is pressed.
- static int event_state ()

  Returns the keyboard and mouse button states of the last event.
- static int event_state (int mask)

  Returns non-zero if any of the passed event state bits are turned on.
- static const char ∗ event_text ()

  Returns the text associated with the current event, including FL_PASTE or FL_DND_RELEASE events.
- static int event_x ()

  Returns the mouse position of the event relative to the Fl_Window it was passed to.
- static int event_x_root ()

  Returns the mouse position on the screen of the event.
- static int event_y ()

  Returns the mouse position of the event relative to the Fl_Window it was passed to.
- static int event_y_root ()

  Returns the mouse position on the screen of the event.
- static Fl_Window ∗ first_window ()

  Returns the first top-level window in the list of shown() windows.
- static void first_window (Fl_Window ∗)

  Sets the window that is returned by first_window().
- static void flush ()

  Causes all the windows that need it to be redrawn and graphics forced out through the pipes.
- static Fl_Widget ∗ focus ()

  Gets the current Fl::focus() widget.
- static void focus (Fl_Widget ∗)

  Sets the widget that will receive FL_KEYBOARD events.
- static void foreground (uchar, uchar, uchar)

  Changes fl_color(FL_FOREGROUND_COLOR).
- static void free_color (Fl_Color i, int overlay=0)

  Frees the specified color from the colormap, if applicable.
- static int get_awake_handler_ (Fl_Awake_Handler &, void ∗&)

  Gets the last stored awake handler for use in awake().
- static Fl_Box_Draw_F ∗ get_boxtype (Fl_Boxtype)

  Gets the current box drawing function for the specified box type.
- static unsigned get_color (Fl_Color i)

  Returns the RGB value(s) for the given FLTK color index.
- static void get_color (Fl_Color i, uchar &red, uchar &green, uchar &blue)

  Returns the RGB value(s) for the given FLTK color index.
- static const char ∗ get_font (Fl_Font)

  Gets the string for this face.
- static const char ∗ get_font_name (Fl_Font, int ∗attributes=0)

Get a human-readable string describing the family of this face.

- static int get_font_sizes (Fl_Font, int ∗&sizep)

  Return an array of sizes in `sizep`*.*
- static int get_key (int key)

  Returns true if the given `key` *is held down now.*
- static void get_mouse (int &, int &)

  Return where the mouse is on the screen by doing a round-trip query to the server.
- static void get_system_colors ()

  Read the user preference colors from the system and use them to call Fl::foreground(), Fl::background(), and Fl::background2().
- static int gl_visual (int, int ∗alist=0)

  This does the same thing as Fl::visual(int) but also requires OpenGL drawing to work.
- static Fl_Window ∗ grab ()

  Returns the window that currently receives all events.
- static void grab (Fl_Window ∗)

  Selects the window to grab.
- static void grab (Fl_Window &win)

  See grab(Fl_Window∗)
- static int h ()

  Returns the height in pixels of the main screen work area.
- static int handle (int, Fl_Window ∗)

  Handle events from the window system.
- static int handle_ (int, Fl_Window ∗)

  Handle events from the window system.
- static int has_check (Fl_Timeout_Handler, void ∗=0)

  Returns 1 if the check exists and has not been called yet, 0 otherwise.
- static int has_idle (Fl_Idle_Handler cb, void ∗data=0)

  Returns true if the specified idle callback is currently installed.
- static int has_timeout (Fl_Timeout_Handler, void ∗=0)

  Returns true if the timeout exists and has not been called yet.
- static int is_scheme (const char ∗name)

  Returns whether the current scheme is the given name.
- static int lock ()

  The lock() method blocks the current thread until it can safely access FLTK widgets and data.
- static Fl_Window ∗ modal ()

  Returns the top-most modal() window currently shown.
- static Fl_Window ∗ next_window (const Fl_Window ∗)

  Returns the next top-level window in the list of shown() windows.
- static bool option (Fl_Option opt)

  FLTK library options management.
- static void option (Fl_Option opt, bool val)

  Override an option while the application is running.
- static void own_colormap ()

  Makes FLTK use its `own colormap`*.*
- static void paste (Fl_Widget &receiver, int source, const char ∗type=Fl::clipboard_plain_text)

  Pastes the data from the selection buffer (`source` *is 0) or the clipboard (*`source` *is 1) into* `receiver`*.*
- static void paste (Fl_Widget &receiver)

Backward compatibility only.
- static Fl_Widget ∗ pushed ()

  Gets the widget that is being pushed.
- static void pushed (Fl_Widget ∗)

  Sets the widget that is being pushed.
- static Fl_Widget ∗ readqueue ()

  Reads the default callback queue and returns the first widget.
- static int ready ()

  This is similar to Fl::check() except this does not call Fl::flush() or any callbacks, which is useful if your program is in a state where such callbacks are illegal.
- static void redraw ()

  Redraws all widgets.
- static void release ()

  Releases the current grabbed window, equals grab(0).
- static void release_widget_pointer (Fl_Widget ∗&w)

  Releases a widget pointer from the watch list.
- static int reload_scheme ()

  Called by scheme according to scheme name.
- static void remove_check (Fl_Timeout_Handler, void ∗=0)

  Removes a check callback.
- static void remove_clipboard_notify (Fl_Clipboard_Notify_Handler h)

  Stop calling the specified callback when there are changes to the selection buffer or the clipboard.
- static void remove_fd (int, int when)

  Removes a file descriptor handler.
- static void remove_fd (int)

  Removes a file descriptor handler.
- static void remove_handler (Fl_Event_Handler h)

  Removes a previously added event handler.
- static void remove_idle (Fl_Idle_Handler cb, void ∗data=0)

  Removes the specified idle callback, if it is installed.
- static void remove_system_handler (Fl_System_Handler h)

  Removes a previously added system event handler.
- static void remove_timeout (Fl_Timeout_Handler, void ∗=0)

  Removes a timeout callback.
- static void repeat_timeout (double t, Fl_Timeout_Handler, void ∗=0)

  Repeats a timeout callback from the expiration of the previous timeout, allowing for more accurate timing.
- static int run ()

  As long as any windows are displayed this calls Fl::wait() repeatedly.
- static int scheme (const char ∗name)

  Sets the current widget scheme.
- static const char ∗ scheme ()

  See void scheme(const char ∗name)
- static int screen_count ()

  Gets the number of available screens.
- static void screen_dpi (float &h, float &v, int n=0)

  Gets the screen resolution in dots-per-inch for the given screen.
- static int screen_num (int x, int y)

Gets the screen number of a screen that contains the specified screen position `x`*,* `y`*.*
- static int screen_num (int x, int y, int w, int h)

  Gets the screen number for the screen which intersects the most with the rectangle defined by `x`*,* `y`*,* `w`*,* `h`*.*
- static void screen_work_area (int &X, int &Y, int &W, int &H, int mx, int my)

  Gets the bounding box of the work area of a screen that contains the specified screen position `mx`*,* `my`*.*
- static void screen_work_area (int &X, int &Y, int &W, int &H, int n)

  Gets the bounding box of the work area of the given screen.
- static void screen_work_area (int &X, int &Y, int &W, int &H)

  Gets the bounding box of the work area of the screen that contains the mouse pointer.
- static void screen_xywh (int &X, int &Y, int &W, int &H)

  Gets the bounding box of a screen that contains the mouse pointer.
- static void screen_xywh (int &X, int &Y, int &W, int &H, int mx, int my)

  Gets the bounding box of a screen that contains the specified screen position `mx`*,* `my`*.*
- static void screen_xywh (int &X, int &Y, int &W, int &H, int n)

  Gets the screen bounding rect for the given screen.
- static void screen_xywh (int &X, int &Y, int &W, int &H, int mx, int my, int mw, int mh)

  Gets the screen bounding rect for the screen which intersects the most with the rectangle defined by `mx`*,* `my`*,* `mw`*,* `mh`*.*
- static int scrollbar_size ()

  Gets the default scrollbar size used by Fl_Browser_, Fl_Help_View, Fl_Scroll, and Fl_Text_Display widgets.
- static void scrollbar_size (int W)

  Sets the default scrollbar size that is used by the Fl_Browser_, Fl_Help_View, Fl_Scroll, and Fl_Text_Display widgets.
- static void selection (Fl_Widget &owner, const char ∗, int len)

  Changes the current selection.
- static Fl_Widget ∗ selection_owner ()

  back-compatibility only: Gets the widget owning the current selection
- static void selection_owner (Fl_Widget ∗)

  Back-compatibility only: The single-argument call can be used to move the selection to another widget or to set the owner to NULL, without changing the actual text of the selection.
- static void set_abort (Fl_Abort_Handler f)

  For back compatibility, sets the void Fl::fatal handler callback.
- static void set_atclose (Fl_Atclose_Handler f)

  For back compatibility, sets the Fl::atclose handler callback.
- static void set_box_color (Fl_Color)

  Sets the drawing color for the box that is currently drawn.
- static void set_boxtype (Fl_Boxtype, Fl_Box_Draw_F ∗, uchar, uchar, uchar, uchar)

  Sets the function to call to draw a specific boxtype.
- static void set_boxtype (Fl_Boxtype, Fl_Boxtype from)

  Copies the from boxtype.
- static void set_color (Fl_Color, uchar, uchar, uchar)

  Sets an entry in the fl_color index table.
- static void set_color (Fl_Color i, unsigned c)

  Sets an entry in the fl_color index table.
- static void set_font (Fl_Font, const char ∗)

  Changes a face.
- static void set_font (Fl_Font, Fl_Font)

Copies one face to another.

- static Fl_Font set_fonts (const char ∗=0)

  FLTK will open the display, and add every fonts on the server to the face table.

- static void set_idle (Fl_Old_Idle_Handler cb)

  Sets an idle callback.

- static void set_labeltype (Fl_Labeltype, Fl_Label_Draw_F ∗, Fl_Label_Measure_F ∗)

  Sets the functions to call to draw and measure a specific labeltype.

- static void set_labeltype (Fl_Labeltype, Fl_Labeltype from)

  Sets the functions to call to draw and measure a specific labeltype.

- static int test_shortcut (Fl_Shortcut)

  Tests the current event, which must be an FL_KEYBOARD or FL_SHORTCUT, against a shortcut value (described in Fl_Button).

- static void ∗ thread_message ()

  The thread_message() method returns the last message that was sent from a child by the awake() method.

- static void unlock ()

  The unlock() method releases the lock that was set using the lock() method.

- static void use_high_res_GL (int val)

  sets whether GL windows should be drawn at high resolution on Apple computers with retina displays

- static int use_high_res_GL ()

  returns whether GL windows should be drawn at high resolution on Apple computers with retina displays.

- static double version ()

  Returns the compiled-in value of the FL_VERSION constant.

- static void visible_focus (int v)

  Gets or sets the visible keyboard focus on buttons and other non-text widgets.

- static int visible_focus ()

  Gets or sets the visible keyboard focus on buttons and other non-text widgets.

- static int visual (int)

  Selects a visual so that your graphics are drawn correctly.

- static int w ()

  Returns the width in pixels of the main screen work area.

- static int wait ()

  Waits until "something happens" and then returns.

- static double wait (double time)

  See int Fl::wait()

- static void watch_widget_pointer (Fl_Widget ∗&w)

  Adds a widget pointer to the widget watch list.

- static int x ()

  Returns the leftmost x coordinate of the main screen work area.

- static int y ()

  Returns the topmost y coordinate of the main screen work area.

Static Public Attributes

- static void(∗ atclose)(Fl_Window ∗, void ∗)

  Back compatibility: default window callback handler.
- static char const ∗const clipboard_image = "image"

  Denotes image data.
- static char const ∗const clipboard_plain_text = "text/plain"

  Denotes plain textual data.
- static void(∗ error)(const char ∗,...) = ::error

  FLTK calls Fl::error() to output a normal error message.
- static void(∗ fatal)(const char ∗,...) = ::fatal

  FLTK calls Fl::fatal() to output a fatal error message.
- static const char ∗const help = helpmsg+13

  Usage string displayed if Fl::args() detects an invalid argument.
- static void(∗ idle)()

  The currently executing idle callback function: DO NOT USE THIS DIRECTLY!
- static void(∗ warning)(const char ∗,...) = ::warning

  FLTK calls Fl::warning() to output a warning message.

31.2.1 Detailed Description

The Fl is the FLTK global (static) class containing state information and global methods for the current application.

31.2.2 Member Enumeration Documentation

enum Fl::Fl_Option

Enumerator for global FLTK options.

These options can be set system wide, per user, or for the running application only.

See Also

Fl::option(Fl_Option, bool)
Fl::option(Fl_Option)

Enumerator

OPTION_ARROW_FOCUS When switched on, moving the text cursor beyond the start or end of a text in a text widget will change focus to the next text widget. (This is considered 'old' behavior)

When switched off (default), the cursor will stop at the end of the text. Pressing Tab or Ctrl-Tab will advance the keyboard focus.

See also: Fl_Input_::tab_nav()

OPTION_VISIBLE_FOCUS If visible focus is switched on (default), FLTK will draw a dotted rectangle inside the widget that will receive the next keystroke. If switched off, no such indicator will be drawn and keyboard navigation is disabled.

OPTION_DND_TEXT If text drag-and-drop is enabled (default), the user can select and drag text from any text widget. If disabled, no dragging is possible, however dropping text from other applications still works.

OPTION_SHOW_TOOLTIPS If tooltips are enabled (default), hovering the mouse over a widget with a tooltip text will open a little tooltip window until the mouse leaves the widget. If disabled, no tooltip is shown.

OPTION_FNFC_USES_GTK When switched on (default), Fl_Native_File_Chooser runs GTK file dialogs if the GTK library is available on the platform (linux/unix only). When switched off, GTK file dialogs aren't used even if the GTK library is available.

OPTION_LAST For internal use only.

31.2.3 Member Function Documentation

static int Fl::abi_check (const int *val* = FL_ABI_VERSION) `[inline]`, `[static]`

Returns whether the runtime library ABI version is correct.

This enables you to check the ABI version of the linked FLTK library at runtime.

Returns 1 (true) if the compiled ABI version (in the header files) and the linked library ABI version (used at runtime) are the same, 0 (false) otherwise.

Argument `val` can be used to query a particular library ABI version. Use for instance 10303 to query if the runtime library is compatible with FLTK ABI version 1.3.3. This is rarely useful.

The default `val` argument is FL_ABI_VERSION, which checks the version defined at configure time (i.e. in the header files at program compilation time) against the linked library version used at runtime. This is particularly useful if you linked with a shared object library, but it also concerns static linking.

See Also

Fl::abi_version()

int Fl::abi_version ( ) `[static]`

Returns the compiled-in value of the FL_ABI_VERSION constant.

This is useful for checking the version of a shared library.

int Fl::add_awake_handler_ (Fl_Awake_Handler *func*, void ∗ *data*) `[static]`

Adds an awake handler for use in awake().

void Fl::add_check (Fl_Timeout_Handler *cb*, void ∗ *argp* = *0*) `[static]`

FLTK will call this callback just before it flushes the display and waits for events.

This is different than an idle callback because it is only called once, then FLTK calls the system and tells it not to return until an event happens.

This can be used by code that wants to monitor the application's state, such as to keep a display up to date. The advantage of using a check callback is that it is called only when no events are pending. If events are coming in quickly, whole blocks of them will be processed before this is called once. This can save significant time and avoid the application falling behind the events.

Sample code:

```
bool state_changed; // anything that changes the display turns this on

void callback(void*) {
 if (!state_changed) return;
 state_changed = false;
 do_expensive_calculation();
 widget-&gt;redraw();
}

main() {
 Fl::add_check(callback);
 return Fl::run();
}
```

static void Fl::add_fd (int *fd*, int *when*, Fl_FD_Handler *cb*, void ∗ = 0) [static]

Adds file descriptor fd to listen to.

When the fd becomes ready for reading Fl::wait() will call the callback and then return. The callback is passed the fd and the arbitrary void∗ argument.

The second version takes a when bitfield, with the bits FL_READ, FL_WRITE, and FL_EXCEPT defined, to indicate when the callback should be done.

There can only be one callback of each type for a file descriptor. Fl::remove_fd() gets rid of *all* the callbacks for a given file descriptor.

Under UNIX *any* file descriptor can be monitored (files, devices, pipes, sockets, etc.). Due to limitations in Microsoft Windows, WIN32 applications can only monitor sockets.

void Fl::add_idle (Fl_Idle_Handler *cb*, void ∗ *data* = 0) [static]

Adds a callback function that is called every time by Fl::wait() and also makes it act as though the timeout is zero (this makes Fl::wait() return immediately, so if it is in a loop it is called repeatedly, and thus the idle fucntion is called repeatedly).

The idle function can be used to get background processing done.

You can have multiple idle callbacks. To remove an idle callback use Fl::remove_idle().

Fl::wait() and Fl::check() call idle callbacks, but Fl::ready() does not.

The idle callback can call any FLTK functions, including Fl::wait(), Fl::check(), and Fl::ready().

FLTK will not recursively call the idle callback.

void Fl::add_timeout (double *t*, Fl_Timeout_Handler *cb*, void ∗ *argp* = 0) [static]

Adds a one-shot timeout callback.

The function will be called by Fl::wait() at *t* seconds after this function is called. The optional void∗ argument is passed to the callback.

You can have multiple timeout callbacks. To remove a timeout callback use Fl::remove_timeout().

If you need more accurate, repeated timeouts, use Fl::repeat_timeout() to reschedule the subsequent timeouts.

The following code will print "TICK" each second on stdout with a fair degree of accuracy:

```
#include <stdio.h>
#include <FL/Fl.H>
#include <FL/Fl_Window.H>
void callback(void*) {
  printf("TICK\n");
  Fl::repeat_timeout(1.0, callback);    // retrigger timeout
}
int main() {
  Fl_Window win(100,100);
  win.show();
  Fl::add_timeout(1.0, callback);       // set up first timeout
  return Fl::run();
}
```

int Fl::api_version ( ) [static]

Returns the compiled-in value of the FL_API_VERSION constant.

This is useful for checking the version of a shared library.

int Fl::arg (int *argc*, char ∗∗ *argv*, int & *i*) [static]

Parse a single switch from `argv`, starting at word `i`.

Returns the number of words eaten (1 or 2, or 0 if it is not recognized) and adds the same value to `i`.

This is the default argument handler used internally by Fl::args(...), but you can use this function if you prefer to step through the standard FLTK switches yourself.

All standard FLTK switches except -bg2 may be abbreviated to just one letter and case is ignored:

- -bg color or -background color

  Sets the background color using Fl::background().

- -bg2 color or -background2 color

  Sets the secondary background color using Fl::background2().

- -display host:n.n

  Sets the X display to use; this option is silently ignored under WIN32 and MacOS.

- -dnd and -nodnd

  Enables or disables drag and drop text operations using Fl::dnd_text_ops().

- -fg color or -foreground color

  Sets the foreground color using Fl::foreground().

- -geometry WxH+X+Y

  Sets the initial window position and size according to the standard X geometry string.

- -iconic

  Iconifies the window using Fl_Window::iconize().

- -kbd and -nokbd

  Enables or disables visible keyboard focus for non-text widgets using Fl::visible_focus().

- -name string

  Sets the window class using Fl_Window::xclass().

- -scheme string

  Sets the widget scheme using Fl::scheme().

- -title string

  Sets the window title using Fl_Window::label().

- -tooltips and -notooltips

  Enables or disables tooltips using Fl_Tooltip::enable().

If your program requires other switches in addition to the standard FLTK options, you will need to pass your own argument handler to Fl::args(int,char∗∗,int&,Fl_Args_Handler) explicitly.

int Fl::args (int *argc*, char ∗∗ *argv*, int & *i*, Fl_Args_Handler *cb* = *0*) `[static]`

Parse command line switches using the `cb` argument handler.

Returns 0 on error, or the number of words processed.

FLTK provides this as an *entirely optional* command line switch parser. You don't have to call it if you don't want to. Everything it can do can be done with other calls to FLTK.

To use the switch parser, call Fl::args(...) near the start of your program. This does **not** open the display, instead switches that need the display open are stashed into static variables. Then you **must** display your first window by calling `window->show(argc,argv)`, which will do anything stored in the static variables.

Providing an argument handler callback `cb` lets you define your own switches. It is called with the same `argc` and `argv`, and with `i` set to the index of the switch to be processed. The `cb` handler should return zero if the switch is unrecognized, and not change `i`. It should return non-zero to indicate the number of words processed if the switch is recognized, i.e. 1 for just the switch, and more than 1 for the switch plus associated parameters. `i` should be incremented by the same amount.

The `cb` handler is called **before** any other tests, so *you can also override any standard FLTK switch* (this is why FLTK can use very short switches instead of the long ones all other toolkits force you to use). See Fl::arg() for descriptions of the standard switches.

On return `i` is set to the index of the first non-switch. This is either:

- The first word that does not start with '-'.
- The word '-' (used by many programs to name stdin as a file)
- The first unrecognized switch (return value is 0).
- `argc`

The return value is `i` unless an unrecognized switch is found, in which case it is zero. If your program takes no arguments other than switches you should produce an error if the return value is less than `argc`.

A usage string is displayed if Fl::args() detects an invalid argument on the command-line. You can change the message by setting the Fl::help pointer.

A very simple command line parser can be found in `examples/howto-parse-args.cxx`

The simpler Fl::args(int argc, char **argv) form is useful if your program does not have command line switches of its own.

void Fl::args (int *argc*, char ** *argv*) `[static]`

Parse all command line switches matching standard FLTK options only.

It parses all the switches, and if any are not recognized it calls Fl::abort(Fl::help), i.e. unlike the long form, an unrecognized switch generates an error message and causes the program to exit.

void Fl::background (uchar *r*, uchar *g*, uchar *b*) `[static]`

Changes fl_color(FL_BACKGROUND_COLOR) to the given color, and changes the gray ramp from 32 to 56 to black to white.

These are the colors used as backgrounds by almost all widgets and used to draw the edges of all the boxtypes.

void Fl::background2 (uchar *r*, uchar *g*, uchar *b*) `[static]`

Changes the alternative background color.

This color is used as a background by Fl_Input and other text widgets.

This call may change fl_color(FL_FOREGROUND_COLOR) if it does not provide sufficient contrast to FL_BACKGROUND2_COLOR.

Fl_Color Fl::box_color (Fl_Color *c*) `[static]`

Gets the drawing color to be used for the background of a box.

This method is only useful inside box drawing code. It returns the color to be used, either fl_inactive(c) if the widget is inactive_r() or `c` otherwise.

int Fl::box_dh (Fl_Boxtype *t*) `[static]`

Returns the height offset for the given boxtype.

See Also

box_dy().

int Fl::box_dw (Fl_Boxtype *t*) `[static]`

Returns the width offset for the given boxtype.

See Also

box_dy().

int Fl::box_dx (Fl_Boxtype *t*) `[static]`

Returns the X offset for the given boxtype.

See Also

box_dy()

int Fl::box_dy (Fl_Boxtype *t*) `[static]`

Returns the Y offset for the given boxtype.

These functions return the offset values necessary for a given boxtype, useful for computing the area inside a box's borders, to prevent overdrawing the borders.

For instance, in the case of a boxtype like FL_DOWN_BOX where the border width might be 2 pixels all around, the above functions would return 2, 2, 4, and 4 for box_dx, box_dy, box_dw, and box_dh respectively.

An example to compute the area inside a widget's box():

```
int X = yourwidget->x() + Fl::box_dx(yourwidget->box());
int Y = yourwidget->y() + Fl::box_dy(yourwidget->box());
int W = yourwidget->w() - Fl::box_dw(yourwidget->box());
int H = yourwidget->h() - Fl::box_dh(yourwidget->box());
```

These functions are mainly useful in the draw() code for deriving custom widgets, where one wants to avoid drawing over the widget's own border box().

int Fl::check ( ) `[static]`

Same as Fl::wait(0).

Calling this during a big calculation will keep the screen up to date and the interface responsive:

```
while (!calculation_done()) {
calculate();
Fl::check();
if (user_hit_abort_button()) break;
}
```

This returns non-zero if any windows are displayed, and 0 if no windows are displayed (this is likely to change in future versions of FLTK).

static int Fl::damage ( ) `[inline]`, `[static]`

If true then flush() will do something.

void Fl::display (const char ∗ *d*) `[static]`

Sets the X display to use for all windows.

Actually this just sets the environment variable $DISPLAY to the passed string, so this only works before you show() the first window or otherwise open the display, and does nothing useful under WIN32.

static void Fl::dnd_text_ops (int *v*) `[inline]`, `[static]`

Gets or sets whether drag and drop text operations are supported.

This specifically affects whether selected text can be dragged from text fields or dragged within a text field as a cut/paste shortcut.

static int Fl::dnd_text_ops ( ) `[inline]`, `[static]`

Gets or sets whether drag and drop text operations are supported.

This specifically affects whether selected text can be dragged from text fields or dragged within a text field as a cut/paste shortcut.

int Fl::draw_box_active ( ) `[static]`

Determines if the currently drawn box is active or inactive.

If inactive, the box color should be changed to the inactive color.

See Also

Fl::box_color(Fl_Color c)

void Fl::flush ( ) `[static]`

Causes all the windows that need it to be redrawn and graphics forced out through the pipes.

This is what wait() does before looking for events.

Note: in multi-threaded applications you should only call Fl::flush() from the main thread. If a child thread needs to trigger a redraw event, it should instead call Fl::awake() to get the main thread to process the event queue.

void Fl::foreground (uchar *r,* uchar *g,* uchar *b*) `[static]`

Changes fl_color(FL_FOREGROUND_COLOR).

int Fl::get_awake_handler_ (Fl_Awake_Handler & *func,* void ∗& *data*) `[static]`

Gets the last stored awake handler for use in awake().

Fl_Box_Draw_F ∗ Fl::get_boxtype (Fl_Boxtype *t*) `[static]`

Gets the current box drawing function for the specified box type.

void Fl::get_system_colors ( ) `[static]`

Read the user preference colors from the system and use them to call Fl::foreground(), Fl::background(), and Fl::background2().

This is done by Fl_Window::show(argc,argv) before applying the -fg and -bg switches.

On X this reads some common values from the Xdefaults database. KDE users can set these values by running the ”krdb” program, and newer versions of KDE set this automatically if you check the ”apply style to other X programs” switch in their control panel.

int Fl::gl_visual (int *mode,* int ∗ *alist* = *0*) `[static]`

This does the same thing as Fl::visual(int) but also requires OpenGL drawing to work.

This *must* be done if you want to draw in normal windows with OpenGL with gl_start() and gl_end(). It may be useful to call this so your X windows use the same visual as an Fl_Gl_Window, which on some servers will reduce colormap flashing.

See Fl_Gl_Window for a list of additional values for the argument.

static int Fl::is_scheme (const char * *name*) [inline], [static]

Returns whether the current scheme is the given name.

This is a fast inline convenience function to support scheme-specific code in widgets, e.g. in their draw() methods, if required.

Use a valid scheme name, not NULL (although NULL is allowed, this is not a useful argument - see below).

If Fl::scheme() has not been set or has been set to the default scheme ("none" or "base"), then this will always return 0 regardless of the argument, because Fl::scheme() is NULL in this case.

Note

> The stored scheme name is always lowercase, and this method will do a case-sensitive compare, so you **must** provide a lowercase string to return the correct value. This is intentional for performance reasons.

Example:

```
if (Fl::is_scheme("gtk+")) { your_code_here(); }
```

Parameters

in	*name*	**lowercase** string of requested scheme name.

Returns

> 1 if the given scheme is active, 0 otherwise.

See Also

> Fl::scheme(const char *name)

bool Fl::option (Fl_Option *opt*) [static]

FLTK library options management.

This function needs to be documented in more detail. It can be used for more optional settings, such as using a native file chooser instead of the FLTK one wherever possible, disabling tooltips, disabling visible focus, disabling FLTK file chooser preview, etc. .

There should be a command line option interface.

There should be an application that manages options system wide, per user, and per application.

Example:

```
if ( Fl::option(Fl::OPTION_ARROW_FOCUS) )
    { ..on.. }
else
    { ..off.. }
```

Note

> As of FLTK 1.3.0, options can be managed within fluid, using the menu *Edit/Global FLTK Settings*.

Parameters

opt	which option

Returns

true or false

See Also

enum Fl::Fl_Option
Fl::option(Fl_Option, bool)

Since

FLTK 1.3.0

void Fl::option (Fl_Option *opt*, bool *val*) [static]

Override an option while the application is running.
This function does not change any system or user settings.
Example:

```
Fl::option(Fl::OPTION_ARROW_FOCUS, true);     // on
Fl::option(Fl::OPTION_ARROW_FOCUS, false);    // off
```

Parameters

opt	which option
val	set to true or false

See Also

enum Fl::Fl_Option
bool Fl::option(Fl_Option)

void Fl::own_colormap ( ) [static]

Makes FLTK use its `own colormap`.

This may make FLTK display better and will reduce conflicts with other programs that want lots of colors. However the colors may flash as you move the cursor between windows.

This does nothing if the current visual is not colormapped.

Fl_Widget ∗ Fl::readqueue ( ) [static]

Reads the default callback queue and returns the first widget.

All Fl_Widgets that don't have a callback defined use the default callback `static` Fl_Widget::default_callback() that puts a pointer to the widget in a queue. This method reads the oldest widget out of this queue.

The queue (FIFO) is limited (currently 20 items). If the queue overflows, the oldest entry (Fl_Widget ∗) is discarded.

Relying on the default callback and reading the callback queue with Fl::readqueue() is not recommended. If you need a callback, you should set one with Fl_Widget::callback(Fl_Callback ∗cb, void ∗data) or one of its variants.

See Also

Fl_Widget::callback()
Fl_Widget::callback(Fl_Callback ∗cb, void ∗data)
Fl_Widget::default_callback()

int Fl::ready ( ) [static]

This is similar to Fl::check() except this does *not* call Fl::flush() or any callbacks, which is useful if your program is in a state where such callbacks are illegal.

This returns true if Fl::check() would do anything (it will continue to return true until you call Fl::check() or Fl::wait()).

```
while (!calculation_done()) {
  calculate();
  if (Fl::ready()) {
    do_expensive_cleanup();
    Fl::check();
    if (user_hit_abort_button()) break;
  }
}
```

static void Fl::release ( ) [inline], [static]

Releases the current grabbed window, equals grab(0).

Deprecated Use Fl::grab(0) instead.

See Also

grab(Fl_Window*)

int Fl::reload_scheme ( ) [static]

Called by scheme according to scheme name.

Loads or reloads the current scheme selection. See void scheme(const char *name)

void Fl::remove_check (Fl_Timeout_Handler *cb*, void * *argp* = 0) [static]

Removes a check callback.

It is harmless to remove a check callback that no longer exists.

static void Fl::remove_fd (int , int *when*) [static]

Removes a file descriptor handler.

static void Fl::remove_fd (int) [static]

Removes a file descriptor handler.

void Fl::remove_timeout (Fl_Timeout_Handler *cb*, void * *argp* = 0) [static]

Removes a timeout callback.

It is harmless to remove a timeout callback that no longer exists.

Note

This version removes all matching timeouts, not just the first one. This may change in the future.

void Fl::repeat_timeout (double *t*, Fl_Timeout_Handler *cb*, void ∗ *argp* = 0) [static]

Repeats a timeout callback from the expiration of the previous timeout, allowing for more accurate timing.

You may only call this method inside a timeout callback.

The following code will print "TICK" each second on stdout with a fair degree of accuracy:

```
void callback(void*) {
  puts("TICK");
  Fl::repeat_timeout(1.0, callback);
}

int main() {
  Fl::add_timeout(1.0, callback);
  return Fl::run();
}
```

int Fl::run ( ) [static]

As long as any windows are displayed this calls Fl::wait() repeatedly.

When all the windows are closed it returns zero (supposedly it would return non-zero on any errors, but FLTK calls exit directly for these). A normal program will end main() with return Fl::run();.

int Fl::scheme (const char ∗ *s*) [static]

Sets the current widget scheme.

NULL will use the scheme defined in the FLTK_SCHEME environment variable or the scheme resource under X11. Otherwise, any of the following schemes can be used:

```
- "none" - This is the default look-n-feel which resembles old
           Windows (95/98/Me/NT/2000) and old GTK/KDE

- "base" - This is an alias for "none"

- "plastic" - This scheme is inspired by the Aqua user interface
              on Mac OS X

- "gtk+" - This scheme is inspired by the Red Hat Bluecurve theme

- "gleam" - This scheme is inspired by the Clearlooks Glossy scheme.
            (Colin Jones and Edmanuel Torres).
```

Uppercase scheme names are equivalent, but the stored scheme name will always be lowercase and Fl::scheme() will return this lowercase name.

If the resulting scheme name is not defined, the default scheme will be used and Fl::scheme() will return NULL.

See Also

Fl::is_scheme()

int Fl::scrollbar_size ( ) [static]

Gets the default scrollbar size used by Fl_Browser_, Fl_Help_View, Fl_Scroll, and Fl_Text_Display widgets.

Returns

The default size for widget scrollbars, in pixels.

void Fl::scrollbar_size (int *W*) [static]

Sets the default scrollbar size that is used by the Fl_Browser_, Fl_Help_View, Fl_Scroll, and Fl_Text_Display widgets.

Parameters

`in`	*W*	The new default size for widget scrollbars, in pixels.

void Fl::set_box_color (Fl_Color *c*) `[static]`

Sets the drawing color for the box that is currently drawn.

This method sets the current drawing color fl_color() depending on the widget's state to either `c` or fl_inactive(c).

It should be used whenever a box background is drawn in the box (type) drawing code instead of calling fl_color(Fl_Color bg) with the background color `bg`, usually Fl_Widget::color().

This method is only useful inside box drawing code. Whenever a box is drawn with one of the standard box drawing methods, a static variable is set depending on the widget's current state - if the widget is inactive_r() then the internal variable is false (0), otherwise it is true (1). This is faster than calling Fl_-Widget::active_r() because the state is cached.

See Also

Fl::draw_box_active()
Fl::box_color(Fl_Color)

void Fl::set_boxtype (Fl_Boxtype *t*, Fl_Box_Draw_F ∗ *f*, uchar *a*, uchar *b*, uchar *c*, uchar *d*) `[static]`

Sets the function to call to draw a specific boxtype.

void Fl::set_boxtype (Fl_Boxtype *to*, Fl_Boxtype *from*) `[static]`

Copies the from boxtype.

static void Fl::set_idle (Fl_Old_Idle_Handler *cb*) `[inline]`, `[static]`

Sets an idle callback.

Deprecated This method is obsolete - use the add_idle() method instead.

void Fl::set_labeltype (Fl_Labeltype *t*, Fl_Label_Draw_F ∗ *f*, Fl_Label_Measure_F ∗ *m*) `[static]`

Sets the functions to call to draw and measure a specific labeltype.

static void Fl::set_labeltype (Fl_Labeltype , Fl_Labeltype *from*) `[static]`

Sets the functions to call to draw and measure a specific labeltype.

static void Fl::use_high_res_GL (int *val*) `[inline]`, `[static]`

sets whether GL windows should be drawn at high resolution on Apple computers with retina displays

Version

1.3.4

static int Fl::use_high_res_GL ( ) `[inline]`, `[static]`

returns whether GL windows should be drawn at high resolution on Apple computers with retina displays.
Default is no.

Version

1.3.4

double Fl::version ( ) `[static]`

Returns the compiled-in value of the FL_VERSION constant.
This is useful for checking the version of a shared library.

Deprecated Use int Fl::api_version() instead.

static void Fl::visible_focus (int *v*) `[inline]`, `[static]`

Gets or sets the visible keyboard focus on buttons and other non-text widgets.
The default mode is to enable keyboard focus for all widgets.

static int Fl::visible_focus ( ) `[inline]`, `[static]`

Gets or sets the visible keyboard focus on buttons and other non-text widgets.
The default mode is to enable keyboard focus for all widgets.

int Fl::visual (int *flags*) `[static]`

Selects a visual so that your graphics are drawn correctly.
This is only allowed before you call show() on any windows. This does nothing if the default visual satisfies the capabilities, or if no visual satisfies the capabilities, or on systems that don't have such brain-dead notions.
Only the following combinations do anything useful:

- Fl::visual(FL_RGB)

  Full/true color (if there are several depths FLTK chooses the largest). Do this if you use fl_draw_-image for much better (non-dithered) output.

- Fl::visual(FL_RGB8)

  Full color with at least 24 bits of color. FL_RGB will always pick this if available, but if not it will happily return a less-than-24 bit deep visual. This call fails if 24 bits are not available.

- Fl::visual(FL_DOUBLE|FL_INDEX)

  Hardware double buffering. Call this if you are going to use Fl_Double_Window.

- Fl::visual(FL_DOUBLE|FL_RGB)
- Fl::visual(FL_DOUBLE|FL_RGB8)

  Hardware double buffering and full color.

This returns true if the system has the capabilities by default or FLTK suceeded in turing them on. Your program will still work even if this returns false (it just won't look as good).

int Fl::wait ( ) `[static]`

Waits until "something happens" and then returns.

Call this repeatedly to "run" your program. You can also check what happened each time after this returns, which is quite useful for managing program state.

What this really does is call all idle callbacks, all elapsed timeouts, call Fl::flush() to get the screen to update, and then wait some time (zero if there are idle callbacks, the shortest of all pending timeouts, or infinity), for any events from the user or any Fl::add_fd() callbacks. It then handles the events and calls the callbacks and then returns.

The return value of Fl::wait() is non-zero if there are any visible windows - this may change in future versions of FLTK.

Fl::wait(time) waits a maximum of *time* seconds. *It can return much sooner if something happens.*

The return value is positive if an event or fd happens before the time elapsed. It is zero if nothing happens (on Win32 this will only return zero if *time* is zero). It is negative if an error occurs (this will happen on UNIX if a signal happens).

31.2.4 Member Data Documentation

const char *const Fl::help = helpmsg+13 `[static]`

Usage string displayed if Fl::args() detects an invalid argument.

This may be changed to point to customized text at run-time.

void(* Fl::idle)() `[static]`

The currently executing idle callback function: DO NOT USE THIS DIRECTLY!

This is now used as part of a higher level system allowing multiple idle callback functions to be called.

See Also

add_idle(), remove_idle()

The documentation for this class was generated from the following files:

- Fl.H
- Fl.cxx
- Fl_abort.cxx
- Fl_add_idle.cxx
- Fl_arg.cxx
- fl_boxtype.cxx
- fl_color.cxx
- fl_color_mac.cxx
- fl_color_win32.cxx
- Fl_compose.cxx
- Fl_display.cxx
- fl_dnd_win32.cxx
- fl_dnd_x.cxx
- Fl_get_key.cxx
- Fl_get_key_mac.cxx
- Fl_get_key_win32.cxx
- Fl_get_system_colors.cxx
- Fl_grab.cxx
- fl_labeltype.cxx
- Fl_lock.cxx
- Fl_own_colormap.cxx

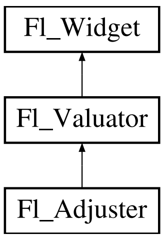
- fl_set_font.cxx
- fl_set_fonts_mac.cxx
- fl_set_fonts_win32.cxx
- fl_set_fonts_x.cxx
- fl_set_fonts_xft.cxx
- fl_shortcut.cxx
- Fl_visual.cxx
- Fl_Widget.cxx
- Fl_Window.cxx
- gl_start.cxx
- screen_xywh.cxx
- Fl_Cairo.cxx

31.3 Fl_Adjuster Class Reference

The Fl_Adjuster widget was stolen from Prisms, and has proven to be very useful for values that need a large dynamic range.

`#include <Fl_Adjuster.H>`

Inheritance diagram for Fl_Adjuster:



Public Member Functions

- Fl_Adjuster (int X, int Y, int W, int H, const char ∗l=0)

  Creates a new Fl_Adjuster widget using the given position, size, and label string.
- void soft (int s)

  If "soft" is turned on, the user is allowed to drag the value outside the range.
- int soft () const

  If "soft" is turned on, the user is allowed to drag the value outside the range.

Protected Member Functions

- void draw ()

  Draws the widget.
- int handle (int)

  Handles the specified event.
- void value_damage ()

  Asks for partial redraw.

Additional Inherited Members

31.3.1 Detailed Description

The Fl_Adjuster widget was stolen from Prisms, and has proven to be very useful for values that need a large dynamic range.

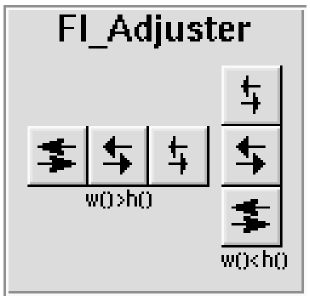


Figure 31.1: Fl_Adjuster

When you press a button and drag to the right the value increases. When you drag to the left it decreases. The largest button adjusts by 100 ∗ step(), the next by 10 ∗ step() and that smallest button by step(). Clicking on the buttons increments by 10 times the amount dragging by a pixel does. Shift + click decrements by 10 times the amount.

31.3.2 Constructor & Destructor Documentation

Fl_Adjuster::Fl_Adjuster (int *X*, int *Y*, int *W*, int *H*, const char ∗ *l* = `0`)

Creates a new Fl_Adjuster widget using the given position, size, and label string.

It looks best if one of the dimensions is 3 times the other.

Inherited destructor destroys the Valuator.

31.3.3 Member Function Documentation

void Fl_Adjuster::draw ( ) `[protected]`, `[virtual]`

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call redraw() instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw() method*, e.g. for an embedded scrollbar, you can do it (because draw() is virtual) like this:

```
Fl_Widget *s = &scroll;              // scroll is an embedded Fl_Scrollbar
s->draw();                   // calls Fl_Scrollbar::draw()
```

Implements Fl_Widget.

int Fl_Adjuster::handle (int *event*) `[protected]`, `[virtual]`

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited handle() method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters

in	*event*	the kind of event received

Return values

0	if the event was not used or understood
1	if the event was used and can be deleted

See Also

Fl_Event

Reimplemented from Fl_Widget.

void Fl_Adjuster::soft (int *s* **)** `[inline]`

If ”soft” is turned on, the user is allowed to drag the value outside the range.

If they drag the value to one of the ends, let go, then grab again and continue to drag, they can get to any value. Default is one.

int Fl_Adjuster::soft ( ) const `[inline]`

If ”soft” is turned on, the user is allowed to drag the value outside the range.

If they drag the value to one of the ends, let go, then grab again and continue to drag, they can get to any value. Default is one.

The documentation for this class was generated from the following files:

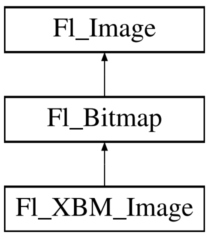
- Fl_Adjuster.H
- Fl_Adjuster.cxx

31.4 Fl_Bitmap Class Reference

The Fl_Bitmap class supports caching and drawing of mono-color (bitmap) images.

`#include <Fl_Bitmap.H>`

Inheritance diagram for Fl_Bitmap:



Public Member Functions

- virtual Fl_Image ∗ copy (int W, int H)

  The copy() method creates a copy of the specified image.
- Fl_Image ∗ **copy** ()
- virtual void draw (int X, int Y, int W, int H, int cx=0, int cy=0)

  Draws the image with a bounding box.
- void **draw** (int X, int Y)
- Fl_Bitmap (const uchar ∗bits, int W, int H)

The constructors create a new bitmap from the specified bitmap data.

- Fl_Bitmap (const char ∗bits, int W, int H)

  The constructors create a new bitmap from the specified bitmap data.
- virtual void label (Fl_Widget ∗w)

  The label() methods are an obsolete way to set the image attribute of a widget or menu item.
- virtual void label (Fl_Menu_Item ∗m)

  The label() methods are an obsolete way to set the image attribute of a widget or menu item.
- virtual void uncache ()

  If the image has been cached for display, delete the cache data.
- virtual ∼Fl_Bitmap ()

  The destructor frees all memory and server resources that are used by the bitmap.

Public Attributes

- int alloc_array

  Non-zero if array points to bitmap data allocated internally.
- const uchar ∗ array

  pointer to raw bitmap data

Friends

- class **Fl_GDI_Graphics_Driver**
- class **Fl_GDI_Printer_Graphics_Driver**
- class **Fl_Quartz_Graphics_Driver**
- class **Fl_Xlib_Graphics_Driver**

Additional Inherited Members

31.4.1 Detailed Description

The Fl_Bitmap class supports caching and drawing of mono-color (bitmap) images.

Images are drawn using the current color.

31.4.2 Constructor & Destructor Documentation

Fl_Bitmap::Fl_Bitmap (const uchar ∗ *bits*, int *W*, int *H*) `[inline]`

The constructors create a new bitmap from the specified bitmap data.

Fl_Bitmap::Fl_Bitmap (const char ∗ *array*, int *W*, int *H*) `[inline]`

The constructors create a new bitmap from the specified bitmap data.

31.4.3 Member Function Documentation

Fl_Image ∗ Fl_Bitmap::copy (int *W*, int *H*) `[virtual]`

The copy() method creates a copy of the specified image.

If the width and height are provided, the image is resized to the specified size. The image should be deleted (or in the case of Fl_Shared_Image, released) when you are done with it.

Reimplemented from Fl_Image.

void Fl_Bitmap::draw (int *X*, int *Y*, int *W*, int *H*, int *cx* = *0*, int *cy* = *0*) `[virtual]`

Draws the image with a bounding box.

Arguments `X,Y,W,H` specify a bounding box for the image, with the origin (upper-left corner) of the image offset by the `cx` and `cy` arguments.

In other words: `fl_push_clip(X,Y,W,H)` is applied, the image is drawn with its upper-left corner at `X-cx,Y-cy` and its own width and height, `fl_pop_clip()` is applied.

Reimplemented from Fl_Image.

void Fl_Bitmap::label (Fl_Widget ∗ *widget*) `[virtual]`

The label() methods are an obsolete way to set the image attribute of a widget or menu item.

Use the image() or deimage() methods of the Fl_Widget and Fl_Menu_Item classes instead.

Reimplemented from Fl_Image.

void Fl_Bitmap::label (Fl_Menu_Item ∗ *m*) `[virtual]`

The label() methods are an obsolete way to set the image attribute of a widget or menu item.

Use the image() or deimage() methods of the Fl_Widget and Fl_Menu_Item classes instead.

Reimplemented from Fl_Image.

void Fl_Bitmap::uncache ( ) `[virtual]`

If the image has been cached for display, delete the cache data.

This allows you to change the data used for the image and then redraw it without recreating an image object.

Reimplemented from Fl_Image.

The documentation for this class was generated from the following files:

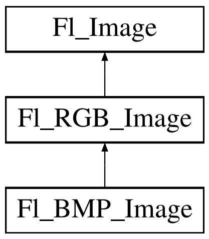
- Fl_Bitmap.H
- Fl_Bitmap.cxx

31.5 Fl_BMP_Image Class Reference

The Fl_BMP_Image class supports loading, caching, and drawing of Windows Bitmap (BMP) image files.

`#include <Fl_BMP_Image.H>`

Inheritance diagram for Fl_BMP_Image:



Public Member Functions

- Fl_BMP_Image (const char ∗filename)

  The constructor loads the named BMP image from the given bmp filename.

Additional Inherited Members

31.5.1 Detailed Description

The Fl_BMP_Image class supports loading, caching, and drawing of Windows Bitmap (BMP) image files.

31.5.2 Constructor & Destructor Documentation

Fl_BMP_Image::Fl_BMP_Image (const char ∗ *bmp*)

The constructor loads the named BMP image from the given bmp filename.

The destructor frees all memory and server resources that are used by the image.

Use Fl_Image::fail() to check if Fl_BMP_Image failed to load. fail() returns ERR_FILE_ACCESS if the file could not be opened or read, ERR_FORMAT if the BMP format could not be decoded, and ERR_NO-_IMAGE if the image could not be loaded for another reason.

The documentation for this class was generated from the following files:

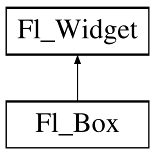
- Fl_BMP_Image.H
- Fl_BMP_Image.cxx

31.6 Fl_Box Class Reference

This widget simply draws its box, and possibly its label.

`#include <Fl_Box.H>`

Inheritance diagram for Fl_Box:



Public Member Functions

- Fl_Box (int X, int Y, int W, int H, const char ∗l=0)
- Fl_Box (Fl_Boxtype b, int X, int Y, int W, int H, const char ∗l)

  See Fl_Box::Fl_Box(int x, int y, int w, int h, const char ∗ = 0)
- virtual int handle (int)

  Handles the specified event.

Protected Member Functions

- void draw ()

  Draws the widget.

Additional Inherited Members

31.6.1 Detailed Description

This widget simply draws its box, and possibly its label.

Putting it before some other widgets and making it big enough to surround them will let you draw a frame around them.

31.6.2 Constructor & Destructor Documentation

Fl_Box::Fl_Box (int *X*, int *Y*, int *W*, int *H*, const char * *l* = 0)

- The first constructor sets box() to FL_NO_BOX, which means it is invisible. However such widgets are useful as placeholders or Fl_Group::resizable() values. To change the box to something visible, use box(n).

- The second form of the constructor sets the box to the specified box type.

The destructor removes the box.

31.6.3 Member Function Documentation

void Fl_Box::draw ( ) `[protected], [virtual]`

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call redraw() instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw() method*, e.g. for an embedded scrollbar, you can do it (because draw() is virtual) like this:

```
Fl_Widget *s = &scroll;              // scroll is an embedded Fl_Scrollbar
s->draw();                   // calls Fl_Scrollbar::draw()
```

Implements Fl_Widget.

int Fl_Box::handle (int *event*) `[virtual]`

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited handle() method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters

`in`	*event*	the kind of event received

Return values

0	if the event was not used or understood
1	if the event was used and can be deleted

See Also

Fl_Event

Reimplemented from Fl_Widget.

The documentation for this class was generated from the following files:

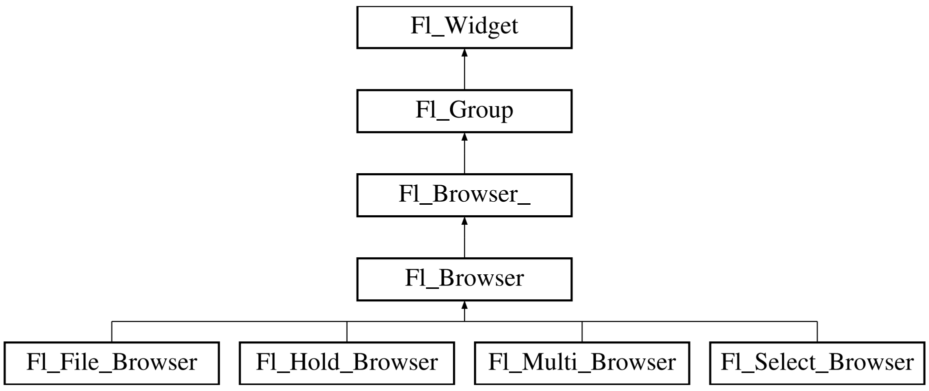
- Fl_Box.H
- Fl_Box.cxx

31.7 Fl_Browser Class Reference

The Fl_Browser widget displays a scrolling list of text lines, and manages all the storage for the text.

`#include <Fl_Browser.H>`

Inheritance diagram for Fl_Browser:



Public Types

- enum Fl_Line_Position { **TOP**, **BOTTOM**, **MIDDLE** }

  For internal use only?

Public Member Functions

- void add (const char ∗newtext, void ∗d=0)

  Adds a new line to the end of the browser.
- void bottomline (int line)

  Scrolls the browser so the bottom item in the browser is showing the specified `line`.
- void clear ()

  Removes all the lines in the browser.
- char column_char () const

  Gets the current column separator character.
- void column_char (char c)

  Sets the column separator to c.
- const int ∗ column_widths () const

  Gets the current column width array.
- void column_widths (const int ∗arr)

  Sets the current array to `arr`.
- void ∗ data (int line) const

  Returns the user data() for specified `line`.
- void data (int line, void ∗d)

  Sets the user data for specified `line` to `d`.
- void display (int line, int val=1)

  For back compatibility.
- int displayed (int line) const

  Returns non-zero if `line` has been scrolled to a position where it is being displayed.
- Fl_Browser (int X, int Y, int W, int H, const char ∗L=0)

The constructor makes an empty browser.

- char format_char () const

  Gets the current format code prefix character, which by default is '@'.
- void format_char (char c)

  Sets the current format code prefix character to `c`*.*
- void hide (int line)

  Makes `line` *invisible, preventing selection by the user.*
- void hide ()

  Hides the entire Fl_Browser widget – opposite of show().
- void icon (int line, Fl_Image ∗icon)

  Set the image icon for `line` *to the value* `icon`*.*
- Fl_Image ∗ icon (int line) const

  Returns the icon currently defined for `line`*.*
- void insert (int line, const char ∗newtext, void ∗d=0)

  Insert a new entry whose label is `newtext` *above given* `line`*, optional data* `d`*.*
- void lineposition (int line, Fl_Line_Position pos)

  Updates the browser so that `line` *is shown at position* `pos`*.*
- int load (const char ∗filename)

  Clears the browser and reads the file, adding each line from the file to the browser.
- void make_visible (int line)

  Make the item at the specified `line` *visible().*
- void middleline (int line)

  Scrolls the browser so the middle item in the browser is showing the specified `line`*.*
- void move (int to, int from)

  Line `from` *is removed and reinserted at* `to`*.*
- void remove (int line)

  Remove entry for given `line` *number, making the browser one line shorter.*
- void remove_icon (int line)

  Removes the icon for `line`*.*
- void replace (int a, const char ∗b)

  For back compatibility only.
- int select (int line, int val=1)

  Sets the selection state of the item at `line` *to the value* `val`*.*
- int selected (int line) const

  Returns 1 if specified `line` *is selected, 0 if not.*
- void show (int line)

  Makes `line` *visible, and available for selection by user.*
- void show ()

  Shows the entire Fl_Browser widget – opposite of hide().
- int size () const

  Returns how many lines are in the browser.
- void **size** (int W, int H)
- void swap (int a, int b)

  Swaps two browser lines `a` *and* `b`*.*
- const char ∗ text (int line) const

  Returns the label text for the specified `line`*.*
- void text (int line, const char ∗newtext)

Sets the text for the specified `line` to `newtext`.

- Fl_Fontsize textsize () const

  Gets the default text size (in pixels) for the lines in the browser.
- void textsize (Fl_Fontsize newSize)

  Sets the default text size (in pixels) for the lines in the browser to `newSize`.
- int topline () const

  Returns the line that is currently visible at the top of the browser.
- void topline (int line)

  Scrolls the browser so the top item in the browser is showing the specified `line`.
- int value () const

  Returns the line number of the currently selected line, or 0 if none selected.
- void value (int line)

  Sets the browser's value(), which selects the specified `line`.
- int visible (int line) const

  Returns non-zero if the specified `line` is visible, 0 if hidden.
- ∼Fl_Browser ()

  The destructor deletes all list items and destroys the browser.

Protected Member Functions

- FL_BLINE ∗ _remove (int line)

  Removes the item at the specified `line`.
- FL_BLINE ∗ find_line (int line) const

  Returns the item for specified `line`.
- int full_height () const

  The height of the entire list of all visible() items in pixels.
- int incr_height () const

  The default 'average' item height (including inter-item spacing) in pixels.
- void insert (int line, FL_BLINE ∗item)

  Insert specified `item` above `line`.
- void ∗ item_at (int line) const

  Return the item at specified `line`.
- void item_draw (void ∗item, int X, int Y, int W, int H) const

  Draws `item` at the position specified by `X Y W H`.
- void ∗ item_first () const

  Returns the very first item in the list.
- int item_height (void ∗item) const

  Returns height of `item` in pixels.
- void ∗ item_last () const

  Returns the very last item in the list.
- void ∗ item_next (void ∗item) const

  Returns the next item after `item`.
- void ∗ item_prev (void ∗item) const

  Returns the previous item before `item`.
- void item_select (void ∗item, int val)

  Change the selection state of `item` to the value `val`.
- int item_selected (void ∗item) const

See if `item` is selected.

- void item_swap (void ∗a, void ∗b)

  Swap the items `a` and `b`.
- const char ∗ item_text (void ∗item) const

  Returns the label text for `item`.
- int item_width (void ∗item) const

  Returns width of `item` in pixels.
- int lineno (void ∗item) const

  Returns line number corresponding to `item`, or zero if not found.
- void swap (FL_BLINE ∗a, FL_BLINE ∗b)

  Swap the two items `a` and `b`.

Additional Inherited Members

31.7.1 Detailed Description

The Fl_Browser widget displays a scrolling list of text lines, and manages all the storage for the text.

This is not a text editor or spreadsheet! But it is useful for showing a vertical list of named objects to the user.

Each line in the browser is identified by number. *The numbers start at one* (this is so that zero can be reserved for "no line" in the selective browsers). *Unless otherwise noted, the methods do not check to see if the passed line number is in range and legal. It must always be greater than zero and <= size().*

Each line contains a null-terminated string of text and a void ∗ data pointer. The text string is displayed, the void ∗ pointer can be used by the callbacks to reference the object the text describes.

The base class does nothing when the user clicks on it. The subclasses Fl_Select_Browser, Fl_Hold_Browser, and Fl_Multi_Browser react to user clicks to select lines in the browser and do callbacks.

The base class Fl_Browser_ provides the scrolling and selection mechanisms of this and all the subclasses, but the dimensions and appearance of each item are determined by the subclass. You can use Fl_Browser_ to display information other than text, or text that is dynamically produced from your own data structures. If you find that loading the browser is a lot of work or is inefficient, you may want to make a subclass of Fl_Browser_.

Some common coding patterns used for working with Fl_Browser:

```
// How to loop through all the items in the browser
for ( int t=1; t<=browser->size(); t++ ) {       // index 1 based..!
    printf("item #%d, label='%s'\n", t, browser->text(t));
}
```

Note: If you are *subclassing* Fl_Browser, it's more efficient to use the protected methods item_first() and item_next(), since Fl_Browser internally uses linked lists to manage the browser's items. For more info, see find_item(int).

31.7.2 Constructor & Destructor Documentation

Fl_Browser::Fl_Browser (int *X*, int *Y*, int *W*, int *H*, const char ∗ *L* = 0)

The constructor makes an empty browser.

Parameters

in	*X,Y,W,H*	position and size.

in	*L*	label string, may be NULL.

31.7.3 Member Function Documentation

FL_BLINE ∗ Fl_Browser::_remove (int *line*) `[protected]`

Removes the item at the specified `line`.

Caveat: See efficiency note in find_line(). You must call redraw() to make any changes visible.

Parameters

in	*line*	The line number to be removed. (1 based) Must be in range!

Returns

Pointer to browser item that was removed (and is no longer valid).

See Also

add(), insert(), remove(), swap(int,int), clear()

void Fl_Browser::add (const char ∗ *newtext*, void ∗ *d* = `0`)

Adds a new line to the end of the browser.

The text string `newtext` may contain format characters; see format_char() for details. `newtext` is copied using the strdup() function, and can be NULL to make a blank line.

The optional void∗ argument `d` will be the data() for the new item.

Parameters

in	*newtext*	The label text used for the added item
in	*d*	Optional user data() for the item (0 if unspecified)

See Also

add(), insert(), remove(), swap(int,int), clear()

void Fl_Browser::bottomline (int *line*) `[inline]`

Scrolls the browser so the bottom item in the browser is showing the specified `line`.

Parameters

in	*line*	The line to be displayed at the bottom.

See Also

topline(), middleline(), bottomline(), displayed(), lineposition()

void Fl_Browser::clear ( )

Removes all the lines in the browser.

See Also

add(), insert(), remove(), swap(int,int), clear()

char Fl_Browser::column_char ( ) const [inline]

Gets the current column separator character.

The default is '\t' (tab).

See Also

column_char(), column_widths()

void Fl_Browser::column_char (char *c*) [inline]

Sets the column separator to c.

This will only have an effect if you also set column_widths(). The default is '\t' (tab).

See Also

column_char(), column_widths()

const int* Fl_Browser::column_widths ( ) const [inline]

Gets the current column width array.

This array is zero-terminated and specifies the widths in pixels of each column. The text is split at each column_char() and each part is formatted into it's own column. After the last column any remaining text is formatted into the space between the last column and the right edge of the browser, even if the text contains instances of column_char() . The default value is a one-element array of just a zero, which means there are no columns.

Example:

```
Fl_Browser *b = new Fl_Browser(..);
static int widths[] = { 50, 50, 50, 70, 70, 40, 40, 70, 70, 50, 0 };  // widths for each column
b->column_widths(widths); // assign array to widget
b->column_char('\t');      // use tab as the column character
b->add("USER\tPID\tCPU\tMEM\tVSZ\tRSS\tTTY\tSTAT\tSTART\tTIME\tCOMMAND");
b->add("root\t2888\t0.0\t0.0\t1352\t0\ttty3\tSW\tAug15\t0:00\t@b@f/sbin/mingetty tty3");
b->add("root\t13115\t0.0\t0.0\t1352\t0\ttty2\tSW\tAug30\t0:00\t@b@f/sbin/mingetty tty2");
[..]
```

See Also

column_char(), column_widths()

void Fl_Browser::column_widths (const int * *arr*) [inline]

Sets the current array to `arr`.

Make sure the last entry is zero.

See Also

column_char(), column_widths()

void * Fl_Browser::data (int *line*) const

Returns the user data() for specified `line`.

Return value can be NULL if `line` is out of range or no user data() was defined. The parameter `line` is 1 based (1 will be the first item in the list).

Parameters

`in`	*line*	The line number of the item whose data() is returned. (1 based)

Returns

The user data pointer (can be NULL)

void Fl_Browser::data (int *line*, void ∗ *d*)

Sets the user data for specified `line` to `d`.

Does nothing if `line` is out of range.

Parameters

`in`	*line*	The line of the item whose data() is to be changed. (1 based)
`in`	*d*	The new data to be assigned to the item. (can be NULL)

void Fl_Browser::display (int *line*, int *val* = 1)

For back compatibility.

This calls show(line) if `val` is true, and hide(line) otherwise. If `val` is not specified, the default is 1 (makes the line visible).

See Also

show(int), hide(int), display(), visible(), make_visible()

int Fl_Browser::displayed (int *line*) const `[inline]`

Returns non-zero if `line` has been scrolled to a position where it is being displayed.

Checks to see if the item's vertical position is within the top and bottom edges of the display window. This does NOT take into account the hide()/show() status of the widget or item.

Parameters

`in`	*line*	The line to be checked

Returns

1 if visible, 0 if not visible.

See Also

topline(), middleline(), bottomline(), displayed(), lineposition()

FL_BLINE ∗ Fl_Browser::find_line (int *line*) const `[protected]`

Returns the item for specified `line`.

Note: This call is slow. It's fine for e.g. responding to user clicks, but slow if called often, such as in a tight sorting loop. Finding an item 'by line' involves a linear lookup on the internal linked list. The performance hit can be significant if the browser's contents is large, and the method is called often (e.g. during a sort). If you're writing a subclass, use the protected methods item_first(), item_next(), etc. to access the internal linked list more efficiently.

Parameters

`in`	*line*	The line number of the item to return. (1 based)

Return values

item	that was found.
NULL	if line is out of range.

See Also

item_at(), find_line(), lineno()

char Fl_Browser::format_char ( ) const `[inline]`

Gets the current format code prefix character, which by default is '@'.

A string of formatting codes at the start of each column are stripped off and used to modify how the rest of the line is printed:

- `'@.'` Print rest of line, don't look for more '@' signs
- `'@@'` Print rest of line starting with '@'
- `'@l'` Use a LARGE (24 point) font
- `'@m'` Use a medium large (18 point) font
- `'@s'` Use a small (11 point) font
- `'@b'` Use a **bold** font (adds FL_BOLD to font)
- `'@i'` Use an *italic* font (adds FL_ITALIC to font)
- `'@f' or '@t'` Use a fixed-pitch font (sets font to FL_COURIER)
- `'@c'` Center the line horizontally
- `'@r'` Right-justify the text
- `'@B0', '@B1', ... '@B255'` Fill the backgound with fl_color(n)
- `'@C0', '@C1', ... '@C255'` Use fl_color(n) to draw the text
- `'@F0', '@F1', ...` Use fl_font(n) to draw the text
- `'@S1', '@S2', ...` Use point size n to draw the text
- `'@u' or '@_'` Underline the text.
- `'@-'` draw an engraved line through the middle.

Notice that the '@.' command can be used to reliably terminate the parsing. To print a random string in a random color, use `sprintf("@C%d@.%s", color, string)` and it will work even if the string starts with a digit or has the format character in it.

void Fl_Browser::format_char (char *c*) `[inline]`

Sets the current format code prefix character to `c`.

The default prefix is '@'. Set the prefix to 0 to disable formatting.

See Also

format_char() for list of '@' codes

int Fl_Browser::full_height ( ) const [protected], [virtual]

The height of the entire list of all visible() items in pixels.

This returns the accumulated height of *all* the items in the browser that are not hidden with hide(), including items scrolled off screen.

Returns

The accumulated size of all the visible items in pixels.

See Also

item_height(), item_width(),
incr_height(), full_height()

Reimplemented from Fl_Browser_.

void Fl_Browser::hide (int *line*)

Makes `line` invisible, preventing selection by the user.

The line can still be selected under program control. This changes the full_height() if the state was changed. When a line is made invisible, lines below it are moved up in the display. redraw() is called automatically if a change occurred.

Parameters

`in`	*line*	The line to be hidden. (1 based)

See Also

show(int), hide(int), display(), visible(), make_visible()

void Fl_Browser::hide ( ) [inline], [virtual]

Hides the entire Fl_Browser widget – opposite of show().

Reimplemented from Fl_Widget.

void Fl_Browser::icon (int *line*, Fl_Image ∗ *icon*)

Set the image icon for `line` to the value `icon`.

Caller is responsible for keeping the icon allocated. The `line` is automatically redrawn.

Parameters

`in`	*line*	The line to be modified. If out of range, nothing is done.
`in`	*icon*	The image icon to be assigned to the `line`. If NULL, any previous icon is removed.

Fl_Image ∗ Fl_Browser::icon (int *line*) const

Returns the icon currently defined for `line`.

If no icon is defined, NULL is returned.

Parameters

in	*line*	The line whose icon is returned.

Returns

The icon defined, or NULL if none.

int Fl_Browser::incr_height ( ) const [protected], [virtual]

The default 'average' item height (including inter-item spacing) in pixels.
This currently returns textsize() + 2.

Returns

The value in pixels.

See Also

item_height(), item_width(),
incr_height(), full_height()

Reimplemented from Fl_Browser_.

void Fl_Browser::insert (int *line*, FL_BLINE ∗ *item*) [protected]

Insert specified `item` above `line`.
If `line` > size() then the line is added to the end.
Caveat: See efficiency note in find_line().
Parameters

in	*line*	The new line will be inserted above this line (1 based).
in	*item*	The item to be added.

void Fl_Browser::insert (int *line*, const char ∗ *newtext*, void ∗ *d* = `0`)

Insert a new entry whose label is `newtext` *above* given `line`, optional data `d`.
Text may contain format characters; see format_char() for details. `newtext` is copied using the strdup() function, and can be NULL to make a blank line.
The optional void ∗ argument `d` will be the data() of the new item.
Parameters

in	*line*	Line position for insert. (1 based) If `line` > size(), the entry will be added at the end.
in	*newtext*	The label text for the new line.
in	*d*	Optional pointer to user data to be associated with the new line.

void∗ Fl_Browser::item_at (int *line*) const [inline], [protected], [virtual]

Return the item at specified `line`.
Parameters

in	*line*	The line of the item to return. (1 based)

Returns

The item, or NULL if line out of range.

See Also

item_at(), find_line(), lineno()

Reimplemented from Fl_Browser_.

void Fl_Browser::item_draw (void * *item*, int *X*, int *Y*, int *W*, int *H*) const [protected], [virtual]

Draws `item` at the position specified by `X Y W H`.

The `W` and `H` values are used for clipping. Should only be called within the context of an FLTK draw().

Parameters

in	*item*	The item to be drawn
in	*X,Y,W,H*	position and size.

Implements Fl_Browser_.

void * Fl_Browser::item_first ( ) const [protected], [virtual]

Returns the very first item in the list.

Example of use:

```
// Walk the browser from beginning to end
for ( void *i=item_first(); i; i=item_next(i) ) {
    printf("item label='%s'\n", item_text(i));
}
```

Returns

The first item, or NULL if list is empty.

See Also

item_first(), item_last(), item_next(), item_prev()

Implements Fl_Browser_.

int Fl_Browser::item_height (void * *item*) const [protected], [virtual]

Returns height of `item` in pixels.

This takes into account embedded @ codes within the text() label.

Parameters

in	*item*	The item whose height is returned.

Returns

The height of the item in pixels.

See Also

item_height(), item_width(),
incr_height(), full_height()

Implements Fl_Browser_.

void * Fl_Browser::item_last ( ) const [protected], [virtual]

Returns the very last item in the list.

Example of use:

```
// Walk the browser in reverse, from end to start
for ( void *i=item_last(); i; i=item_prev(i) ) {
    printf("item label='%s'\n", item_text(i));
}
```

Returns

The last item, or NULL if list is empty.

See Also

item_first(), item_last(), item_next(), item_prev()

Reimplemented from Fl_Browser_.

void * Fl_Browser::item_next (void * *item*) const [protected], [virtual]

Returns the next item after `item`.

Parameters

in	*item*	The 'current' item

Returns

The next item after `item`, or NULL if there are none after this one.

See Also

item_first(), item_last(), item_next(), item_prev()

Implements Fl_Browser_.

void * Fl_Browser::item_prev (void * *item*) const [protected], [virtual]

Returns the previous item before `item`.

Parameters

in	*item*	The 'current' item

Returns

The previous item before `item`, or NULL if there are none before this one.

See Also

item_first(), item_last(), item_next(), item_prev()

Implements Fl_Browser_.

void Fl_Browser::item_select (void * *item*, int *val*) [protected], [virtual]

Change the selection state of `item` to the value `val`.

Parameters

`in`	*item*	The item to be changed.
`in`	*val*	The new selection state: 1 selects, 0 de-selects.

See Also

select(), selected(), value(), item_select(), item_selected()

Reimplemented from Fl_Browser_.

int Fl_Browser::item_selected (void ∗ *item*) const `[protected], [virtual]`

See if `item` is selected.
Parameters

`in`	*item*	The item whose selection state is to be checked.

Returns

1 if selected, 0 if not.

See Also

select(), selected(), value(), item_select(), item_selected()

Reimplemented from Fl_Browser_.

void Fl_Browser::item_swap (void ∗ *a*, void ∗ *b*) `[inline], [protected], [virtual]`

Swap the items `a` and `b`.
You must call redraw() to make any changes visible.
Parameters

`in`	*a,b*	the items to be swapped.

See Also

swap(int,int), item_swap()

Reimplemented from Fl_Browser_.

const char ∗ Fl_Browser::item_text (void ∗ *item*) const `[protected], [virtual]`

Returns the label text for `item`.
Parameters

`in`	*item*	The item whose label text is returned.

Returns

The item's text string. (Can be NULL)

Reimplemented from Fl_Browser_.

int Fl_Browser::item_width (void ∗ *item*) const `[protected], [virtual]`

Returns width of `item` in pixels.
This takes into account embedded @ codes within the text() label.

Parameters

`in`	*item*	The item whose width is returned.

Returns

The width of the item in pixels.

See Also

item_height(), item_width(),
incr_height(), full_height()

Implements Fl_Browser_.

int Fl_Browser::lineno (void ∗ *item*) const `[protected]`

Returns line number corresponding to `item`, or zero if not found.
Caveat: See efficiency note in find_line().
Parameters

`in`	*item*	The item to be found

Returns

The line number of the item, or 0 if not found.

See Also

item_at(), find_line(), lineno()

void Fl_Browser::lineposition (int *line*, Fl_Line_Position *pos*)

Updates the browser so that `line` is shown at position `pos`.
Parameters

`in`	*line*	line number. (1 based)
`in`	*pos*	position.

See Also

topline(), middleline(), bottomline()

int Fl_Browser::load (const char ∗ *filename*)

Clears the browser and reads the file, adding each line from the file to the browser.
If the filename is NULL or a zero-length string then this just clears the browser. This returns zero if there was any error in opening or reading the file, in which case errno is set to the system error. The data() of each line is set to NULL.
Parameters

in	*filename*	The filename to load

Returns

1 if OK, 0 on error (errno has reason)

See Also

add()

void Fl_Browser::make_visible (int *line*) `[inline]`

Make the item at the specified `line` visible().

Functionally similar to show(int line). If `line` is out of range, redisplay top or bottom of list as appropriate.

Parameters

in	*line*	The line to be made visible.

See Also

show(int), hide(int), display(), visible(), make_visible()

void Fl_Browser::middleline (int *line*) `[inline]`

Scrolls the browser so the middle item in the browser is showing the specified `line`.

Parameters

in	*line*	The line to be displayed in the middle.

See Also

topline(), middleline(), bottomline(), displayed(), lineposition()

void Fl_Browser::move (int *to*, int *from*)

Line `from` is removed and reinserted at `to`.

Note: `to` is calculated *after* line `from` gets removed.

Parameters

in	*to*	Destination line number (calculated *after* line `from` is removed)
in	*from*	Line number of item to be moved

void Fl_Browser::remove (int *line*)

Remove entry for given `line` number, making the browser one line shorter.

You must call redraw() to make any changes visible.

Parameters

in	*line*	Line to be removed. (1 based) If `line` is out of range, no action is taken.

See Also

add(), insert(), remove(), swap(int,int), clear()

void Fl_Browser::remove_icon (int *line*)

Removes the icon for `line`.

It's ok to remove an icon if none has been defined.

Parameters

in	*line*	The line whose icon is to be removed.

void Fl_Browser::replace (int *a*, const char ∗ *b*) `[inline]`

For back compatibility only.

int Fl_Browser::select (int *line*, int *val* = `1`)

Sets the selection state of the item at `line` to the value `val`.

If `val` is not specified, the default is 1 (selects the item).

Parameters

in	*line*	The line number of the item to be changed. (1 based)
in	*val*	The new selection state (1=select, 0=de-select).

Returns

1 if the state changed, 0 if not.

See Also

select(), selected(), value(), item_select(), item_selected()

int Fl_Browser::selected (int *line*) const

Returns 1 if specified `line` is selected, 0 if not.

Parameters

in	*line*	The line being checked (1 based)

Returns

1 if item selected, 0 if not.

See Also

select(), selected(), value(), item_select(), item_selected()

void Fl_Browser::show (int *line*)

Makes `line` visible, and available for selection by user.

Opposite of hide(int). This changes the full_height() if the state was changed. redraw() is called automatically if a change occurred.

Parameters

in	*line*	The line to be shown. (1 based)

See Also

show(int), hide(int), display(), visible(), make_visible()

void Fl_Browser::show ( ) [inline], [virtual]

Shows the entire Fl_Browser widget – opposite of hide().
Reimplemented from Fl_Widget.

int Fl_Browser::size ( ) const [inline]

Returns how many lines are in the browser.
The last line number is equal to this. Returns 0 if browser is empty.

void Fl_Browser::swap (FL_BLINE ∗ *a*, FL_BLINE ∗ *b*) [protected]

Swap the two items `a` and `b`.
Uses swapping() to ensure list updates correctly.
Parameters

in	*a,b*	The two items to be swapped.

See Also

swap(int,int), item_swap()

void Fl_Browser::swap (int *a*, int *b*)

Swaps two browser lines `a` and `b`.
You must call redraw() to make any changes visible.
Parameters

in	*a,b*	The two lines to be swapped. (both 1 based)

See Also

swap(int,int), item_swap()

const char ∗ Fl_Browser::text (int *line*) const

Returns the label text for the specified `line`.
Return value can be NULL if `line` is out of range or unset. The parameter `line` is 1 based.
Parameters

in	*line*	The line number of the item whose text is returned. (1 based)

Returns

The text string (can be NULL)

void Fl_Browser::text (int *line,* const char ∗ *newtext*)

Sets the text for the specified `line` to `newtext`.

Text may contain format characters; see format_char() for details. `newtext` is copied using the strdup() function, and can be NULL to make a blank line.

Does nothing if `line` is out of range.

Parameters

`in`	*line*	The line of the item whose text will be changed. (1 based)
`in`	*newtext*	The new string to be assigned to the item.

void Fl_Browser::textsize (Fl_Fontsize *newSize*)

Sets the default text size (in pixels) for the lines in the browser to `newSize`.

This method recalculates all item heights and caches the total height internally for optimization of later item changes. This can be slow if there are many items in the browser.

It returns immediately (w/o recalculation) if `newSize` equals the current textsize().

You *may* need to call redraw() to see the effect and to have the scrollbar positions recalculated.

You should set the text size *before* populating the browser with items unless you really need to *change* the size later.

int Fl_Browser::topline ( ) const

Returns the line that is currently visible at the top of the browser.

If there is no vertical scrollbar then this will always return 1.

Returns

The lineno() of the top() of the browser.

void Fl_Browser::topline (int *line*) `[inline]`

Scrolls the browser so the top item in the browser is showing the specified `line`.

Parameters

`in`	*line*	The line to be displayed at the top.

See Also

topline(), middleline(), bottomline(), displayed(), lineposition()

int Fl_Browser::value ( ) const

Returns the line number of the currently selected line, or 0 if none selected.

Returns

The line number of current selection, or 0 if none selected.

See Also

select(), selected(), value(), item_select(), item_selected()

void Fl_Browser::value (int *line*) `[inline]`

Sets the browser's value(), which selects the specified `line`.

This is the same as calling select(line).

See Also

select(), selected(), value(), item_select(), item_selected()

int Fl_Browser::visible (int *line*) const

Returns non-zero if the specified `line` is visible, 0 if hidden.

Use show(int), hide(int), or make_visible(int) to change an item's visible state.

Parameters

`in`	*line*	The line in the browser to be tested. (1 based)

See Also

show(int), hide(int), display(), visible(), make_visible()

The documentation for this class was generated from the following files:

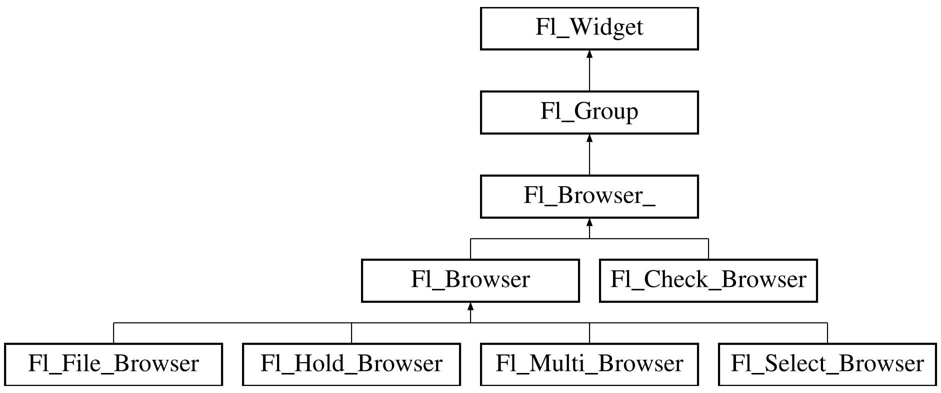
- Fl_Browser.H
- Fl_Browser.cxx
- Fl_Browser_load.cxx

31.8 Fl_Browser_ Class Reference

This is the base class for browsers.

`#include <Fl_Browser_.H>`

Inheritance diagram for Fl_Browser_:



Public Types

- enum {
  HORIZONTAL = 1, VERTICAL = 2, BOTH = 3, ALWAYS_ON = 4,
  HORIZONTAL_ALWAYS = 5, VERTICAL_ALWAYS = 6, BOTH_ALWAYS = 7 }

  Values for has_scrollbar().

Public Member Functions

- int deselect (int docallbacks=0)

  Deselects all items in the list and returns 1 if the state changed or 0 if it did not.
- void display (void ∗item)

  Displays the `item`*, scrolling the list as necessary.*
- int handle (int event)

  Handles the `event` *within the normal widget bounding box.*
- uchar has_scrollbar () const

  Returns the current scrollbar mode, see Fl_Browser_::has_scrollbar(uchar)
- void has_scrollbar (uchar mode)

  Sets whether the widget should have scrollbars or not (default Fl_Browser_::BOTH).
- int hposition () const

  Gets the horizontal scroll position of the list as a pixel position `pos`*.*
- void hposition (int)

  Sets the horizontal scroll position of the list to pixel position `pos`*.*
- int position () const

  Gets the vertical scroll position of the list as a pixel position `pos`*.*
- void position (int pos)

  Sets the vertical scroll position of the list to pixel position `pos`*.*
- void resize (int X, int Y, int W, int H)

  Repositions and/or resizes the browser.
- void scrollbar_left ()

  Moves the vertical scrollbar to the lefthand side of the list.
- void scrollbar_right ()

  Moves the vertical scrollbar to the righthand side of the list.
- int scrollbar_size () const

  Gets the current size of the scrollbars' troughs, in pixels.
- void scrollbar_size (int newSize)

  Sets the pixel size of the scrollbars' troughs to `newSize`*, in pixels.*
- int scrollbar_width () const

  This method has been deprecated, existing for backwards compatibility only.
- void scrollbar_width (int width)

  This method has been deprecated, existing for backwards compatibility only.
- int select (void ∗item, int val=1, int docallbacks=0)

  Sets the selection state of `item` *to* `val`*, and returns 1 if the state changed or 0 if it did not.*
- int select_only (void ∗item, int docallbacks=0)

  Selects `item` *and returns 1 if the state changed or 0 if it did not.*
- void sort (int flags=0)

  Sort the items in the browser based on `flags`*.*
- Fl_Color textcolor () const

  Gets the default text color for the lines in the browser.
- void textcolor (Fl_Color col)

  Sets the default text color for the lines in the browser to color `col`*.*
- Fl_Font textfont () const

  Gets the default text font for the lines in the browser.
- void textfont (Fl_Font font)

Sets the default text font for the lines in the browser to `font`*.*
- Fl_Fontsize textsize () const

  Gets the default text size (in pixels) for the lines in the browser.
- void textsize (Fl_Fontsize newSize)

  Sets the default text size (in pixels) for the lines in the browser to `size`*.*

Public Attributes

- Fl_Scrollbar hscrollbar

  Horizontal scrollbar.
- Fl_Scrollbar scrollbar

  Vertical scrollbar.

Protected Member Functions

- void bbox (int &X, int &Y, int &W, int &H) const

  Returns the bounding box for the interior of the list's display window, inside the scrollbars.
- void deleting (void ∗item)

  This method should be used when `item` *is being deleted from the list.*
- int displayed (void ∗item) const

  Returns non-zero if `item` *has been scrolled to a position where it is being displayed.*
- void draw ()

  Draws the list within the normal widget bounding box.
- void ∗ find_item (int ypos)

  This method returns the item under mouse y position `ypos`*.*
- Fl_Browser_ (int X, int Y, int W, int H, const char ∗L=0)

  The constructor makes an empty browser.
- virtual int full_height () const

  This method may be provided by the subclass to indicate the full height of the item list, in pixels.
- virtual int full_width () const

  This method may be provided by the subclass to indicate the full width of the item list, in pixels.
- virtual int incr_height () const

  This method may be provided to return the average height of all items to be used for scrolling.
- void inserting (void ∗a, void ∗b)

  This method should be used when an item is in the process of being inserted into the list.
- virtual void ∗ item_at (int index) const

  This method must be provided by the subclass to return the item for the specified `index`*.*
- virtual void item_draw (void ∗item, int X, int Y, int W, int H) const =0

  This method must be provided by the subclass to draw the `item` *in the area indicated by* `X`, `Y`, `W`, `H`*.*
- virtual void ∗ item_first () const =0

  This method must be provided by the subclass to return the first item in the list.
- virtual int item_height (void ∗item) const =0

  This method must be provided by the subclass to return the height of `item` *in pixels.*
- virtual void ∗ item_last () const

  This method must be provided by the subclass to return the last item in the list.
- virtual void ∗ item_next (void ∗item) const =0

  This method must be provided by the subclass to return the item in the list after `item`*.*
- virtual void ∗ item_prev (void ∗item) const =0

  This method must be provided by the subclass to return the item in the list before `item`*.*
- virtual int item_quick_height (void ∗item) const

  This method may be provided by the subclass to return the height of the `item`*, in pixels.*
- virtual void item_select (void ∗item, int val=1)

  This method must be implemented by the subclass if it supports multiple selections; sets the selection state to `val` *for the* `item`*.*
- virtual int item_selected (void ∗item) const

  This method must be implemented by the subclass if it supports multiple selections; returns the selection state for `item`*.*
- virtual void item_swap (void ∗a, void ∗b)

  This optional method should be provided by the subclass to efficiently swap browser items `a` *and* `b`*, such as for sorting.*
- virtual const char ∗ item_text (void ∗item) const

  This optional method returns a string (label) that may be used for sorting.
- virtual int item_width (void ∗item) const =0

  This method must be provided by the subclass to return the width of the `item` *in pixels.*
- int leftedge () const

  This method returns the X position of the left edge of the list area after adjusting for the scrollbar and border, if any.
- void new_list ()

  This method should be called when the list data is completely replaced or cleared.
- void redraw_line (void ∗item)

  This method should be called when the contents of `item` *has changed, but not its height.*
- void redraw_lines ()

  This method will cause the entire list to be redrawn.
- void replacing (void ∗a, void ∗b)

  This method should be used when item `a` *is being replaced by item* `b`*.*
- void ∗ selection () const

  Returns the item currently selected, or NULL if there is no selection.
- void swapping (void ∗a, void ∗b)

  This method should be used when two items `a` *and* `b` *are being swapped.*
- void ∗ top () const

  Returns the item that appears at the top of the list.

Additional Inherited Members

31.8.1 Detailed Description

This is the base class for browsers.

To be useful it must be subclassed and several virtual functions defined. The Forms-compatible browser and the file chooser's browser are subclassed off of this.

This has been designed so that the subclass has complete control over the storage of the data, although because next() and prev() functions are used to index, it works best as a linked list or as a large block of characters in which the line breaks must be searched for.

A great deal of work has been done so that the "height" of a data object does not need to be determined until it is drawn. This is useful if actually figuring out the size of an object requires accessing image data or doing stat() on a file or doing some other slow operation.

Keyboard navigation of browser items

The keyboard navigation of browser items is only possible if visible_focus() is enabled. If disabled, the widget rejects keyboard focus; Tab and Shift-Tab focus navigation will skip the widget.

In 'Select' and 'Normal' mode, the widget rejects keyboard focus; no navigation keys are supported (other than scrollbar positioning).

In 'Hold' mode, the widget accepts keyboard focus, and Up/Down arrow keys can navigate the selected item.

In 'Multi' mode, the widget accepts keyboard focus, and Up/Down arrow keys navigate the focus box; Space toggles the current item's selection, Enter selects only the current item (deselects all others). If Shift (or Ctrl) is combined with Up/Down arrow keys, the current item's selection state is extended to the next item. In this way one can extend a selection or de-selection.

31.8.2 Member Enumeration Documentation

anonymous enum

Values for has_scrollbar().

Anonymous enum bit flags for has_scrollbar().

- bit 0: horizontal
- bit 1: vertical
- bit 2: 'always' (to be combined with bits 0 and 1)
- bit 3-31: reserved for future use

Enumerator

HORIZONTAL Only show horizontal scrollbar.

VERTICAL Only show vertical scrollbar.

BOTH Show both scrollbars. (default)

ALWAYS_ON Specified scrollbar(s) should 'always' be shown (to be used with HORIZONTAL/VERTICAL)

HORIZONTAL_ALWAYS Horizontal scrollbar always on.

VERTICAL_ALWAYS Vertical scrollbar always on.

BOTH_ALWAYS Both scrollbars always on.

31.8.3 Constructor & Destructor Documentation

Fl_Browser_::Fl_Browser_ (int *X*, int *Y*, int *W*, int *H*, const char ∗ *L* = *0*) `[protected]`

The constructor makes an empty browser.

Parameters

`in`	*X,Y,W,H*	position and size.
`in`	*L*	The label string, may be NULL.

31.8.4 Member Function Documentation

void Fl_Browser_::bbox (int & *X*, int & *Y*, int & *W*, int & *H*) const `[protected]`

Returns the bounding box for the interior of the list's display window, inside the scrollbars.

Parameters

`out`	*X,Y,W,H*	The returned bounding box. (The original contents of these parameters are overwritten)

void Fl_Browser_::deleting (void ∗ *item*) `[protected]`

This method should be used when `item` is being deleted from the list.

It allows the Fl_Browser_ to discard any cached data it has on the item. This method does not actually delete the item, but handles the follow up bookkeeping after the item has just been deleted.

Parameters

`in`	*item*	The item being deleted.

int Fl_Browser_::deselect (int *docallbacks* = `0`)

Deselects all items in the list and returns 1 if the state changed or 0 if it did not.

If the optional `docallbacks` parameter is non-zero, deselect tries to call the callback function for the widget.

Parameters

`in`	*docallbacks*	If 1, invokes widget callback if item changed. If 0, doesn't do callback (default).

void Fl_Browser_::display (void ∗ *item*)

Displays the `item`, scrolling the list as necessary.

Parameters

`in`	*item*	The item to be displayed.

See Also

display(), displayed()

int Fl_Browser_::displayed (void ∗ *item*) const `[protected]`

Returns non-zero if `item` has been scrolled to a position where it is being displayed.

Checks to see if the item's vertical position is within the top and bottom edges of the display window. This does NOT take into account the hide()/show() status of the widget or item.

Parameters

`in`	*item*	The item to check

Returns

1 if visible, 0 if not visible.

See Also

display(), displayed()

void ∗ Fl_Browser_::find_item (int *ypos*) `[protected]`

This method returns the item under mouse y position `ypos`.

NULL is returned if no item is displayed at that position.

Parameters

in	*ypos*	The y position (eg. Fl::event_y()) to find an item under.

Returns

The item, or NULL if not found

int Fl_Browser_::full_height ( ) const `[protected]`, `[virtual]`

This method may be provided by the subclass to indicate the full height of the item list, in pixels.

The default implementation computes the full height from the item heights. Includes the items that are scrolled off screen.

Returns

The height of the entire list, in pixels.

Reimplemented in Fl_Browser.

int Fl_Browser_::full_width ( ) const `[protected]`, `[virtual]`

This method may be provided by the subclass to indicate the full width of the item list, in pixels.

The default implementation computes the full width from the item widths.

Returns

The maximum width of all the items, in pixels.

int Fl_Browser_::handle (int *event*) `[virtual]`

Handles the `event` within the normal widget bounding box.

Parameters

in	*event*	The event to process.

Returns

1 if event was processed, 0 if not.

Reimplemented from Fl_Widget.
Reimplemented in Fl_Check_Browser.

void Fl_Browser_::has_scrollbar (uchar *mode*) `[inline]`

Sets whether the widget should have scrollbars or not (default Fl_Browser_::BOTH).

By default you can scroll in both directions, and the scrollbars disappear if the data will fit in the widget. has_scrollbar() changes this based on the value of `mode`:

- 0 - No scrollbars.
- Fl_Browser_::HORIZONTAL - Only a horizontal scrollbar.
- Fl_Browser_::VERTICAL - Only a vertical scrollbar.
- Fl_Browser_::BOTH - The default is both scrollbars.
- Fl_Browser_::HORIZONTAL_ALWAYS - Horizontal scrollbar always on, vertical always off.
- Fl_Browser_::VERTICAL_ALWAYS - Vertical scrollbar always on, horizontal always off.
- Fl_Browser_::BOTH_ALWAYS - Both always on.

int Fl_Browser_::hposition ( ) const `[inline]`

Gets the horizontal scroll position of the list as a pixel position `pos`.

The position returned is how many pixels of the list are scrolled off the left edge of the screen. Example: A position of '18' indicates the left 18 pixels of the list are scrolled off the left edge of the screen.

See Also

position(), hposition()

void Fl_Browser_::hposition (int *pos*)

Sets the horizontal scroll position of the list to pixel position `pos`.

The position is how many pixels of the list are scrolled off the left edge of the screen. Example: A position of '18' scrolls the left 18 pixels of the list off the left edge of the screen.

Parameters

`in`	*pos*	The horizontal position (in pixels) to scroll the browser to.

See Also

position(), hposition()

int Fl_Browser_::incr_height ( ) const `[protected], [virtual]`

This method may be provided to return the average height of all items to be used for scrolling.

The default implementation uses the height of the first item.

Returns

The average height of items, in pixels.

Reimplemented in Fl_Browser.

void Fl_Browser_::inserting (void ∗ *a*, void ∗ *b*) `[protected]`

This method should be used when an item is in the process of being inserted into the list.

It allows the Fl_Browser_ to update its cache data as needed, scheduling a redraw for the affected lines. This method does not actually insert items, but handles the follow up bookkeeping after items have been inserted.

Parameters

`in`	*a*	The starting item position
`in`	*b*	The new item being inserted

virtual void∗ Fl_Browser_::item_at (int *index*) const `[inline], [protected], [virtual]`

This method must be provided by the subclass to return the item for the specified `index`.

Parameters

`in`	*index*	The `index` of the item to be returned

Returns

The item at the specified `index`.

Reimplemented in Fl_Browser.

virtual void∗ Fl_Browser_::item_first ( ) const `[protected], [pure virtual]`

This method must be provided by the subclass to return the first item in the list.

See Also

item_first(), item_next(), item_last(), item_prev()

Implemented in Fl_Browser.

virtual int Fl_Browser_::item_height (void ∗ *item*) const `[protected], [pure virtual]`

This method must be provided by the subclass to return the height of `item` in pixels.
Allow for two additional pixels for the list selection box.
Parameters

`in`	*item*	The item whose height is returned.

Returns

The height of the specified `item` in pixels.

See Also

item_height(), item_width(), item_quick_height()

Implemented in Fl_Browser.

virtual void∗ Fl_Browser_::item_last ( ) const `[inline], [protected], [virtual]`

This method must be provided by the subclass to return the last item in the list.

See Also

item_first(), item_next(), item_last(), item_prev()

Reimplemented in Fl_Browser.

virtual void∗ Fl_Browser_::item_next (void ∗ *item*) const `[protected], [pure virtual]`

This method must be provided by the subclass to return the item in the list after `item`.

See Also

item_first(), item_next(), item_last(), item_prev()

Implemented in Fl_Browser.

virtual void∗ Fl_Browser_::item_prev (void ∗ *item*) const `[protected], [pure virtual]`

This method must be provided by the subclass to return the item in the list before `item`.

See Also

item_first(), item_next(), item_last(), item_prev()

Implemented in Fl_Browser.

int Fl_Browser_::item_quick_height (void ∗ *item*) const `[protected], [virtual]`

This method may be provided by the subclass to return the height of the `item`, in pixels.
Allow for two additional pixels for the list selection box. This method differs from item_height in that it is only called for selection and scrolling operations. The default implementation calls item_height.

Parameters

in	*item*	The item whose height to return.

Returns

The height, in pixels.

void Fl_Browser_::item_select (void ∗ *item*, int *val* = *1*) `[protected]`, `[virtual]`

This method must be implemented by the subclass if it supports multiple selections; sets the selection state to `val` for the `item`.

Sets the selection state for `item`, where optional `val` is 1 (select, the default) or 0 (de-select).

Parameters

in	*item*	The item to be selected
in	*val*	The optional selection state; 1=select, 0=de-select. The default is to select the item (1).

Reimplemented in Fl_Browser.

int Fl_Browser_::item_selected (void ∗ *item*) const `[protected]`, `[virtual]`

This method must be implemented by the subclass if it supports multiple selections; returns the selection state for `item`.

The method should return 1 if `item` is selected, or 0 otherwise.

Parameters

in	*item*	The item to test.

Reimplemented in Fl_Browser.

virtual void Fl_Browser_::item_swap (void ∗ *a*, void ∗ *b*) `[inline]`, `[protected]`, `[virtual]`

This optional method should be provided by the subclass to efficiently swap browser items `a` and `b`, such as for sorting.

Parameters

in	*a,b*	The two items to be swapped.

Reimplemented in Fl_Browser.

virtual const char∗ Fl_Browser_::item_text (void ∗ *item*) const `[inline]`, `[protected]`, `[virtual]`

This optional method returns a string (label) that may be used for sorting.

Parameters

in	*item*	The item whose label text is returned.

Returns

The item's text label. (Can be NULL if blank)

Reimplemented in Fl_Browser.

virtual int Fl_Browser_::item_width (void ∗ *item*) const `[protected]`, `[pure virtual]`

This method must be provided by the subclass to return the width of the `item` in pixels.

Allow for two additional pixels for the list selection box.

Parameters

in	*item*	The item whose width is returned.

Returns

The width of the item in pixels.

Implemented in Fl_Browser.

int Fl_Browser_::leftedge ( ) const `[protected]`

This method returns the X position of the left edge of the list area after adjusting for the scrollbar and border, if any.

Returns

The X position of the left edge of the list, in pixels.

See Also

Fl_Browser_::bbox()

void Fl_Browser_::new_list ( ) `[protected]`

This method should be called when the list data is completely replaced or cleared.

It informs the Fl_Browser_ widget that any cached information it has concerning the items is invalid. This method does not clear the list, it just handles the follow up bookkeeping after the list has been cleared.

int Fl_Browser_::position ( ) const `[inline]`

Gets the vertical scroll position of the list as a pixel position `pos`.

The position returned is how many pixels of the list are scrolled off the top edge of the screen. Example: A position of '3' indicates the top 3 pixels of the list are scrolled off the top edge of the screen.

See Also

position(), hposition()

void Fl_Browser_::position (int *pos*)

Sets the vertical scroll position of the list to pixel position `pos`.

The position is how many pixels of the list are scrolled off the top edge of the screen. Example: A position of '3' scrolls the top three pixels of the list off the top edge of the screen.

Parameters

in	*pos*	The vertical position (in pixels) to scroll the browser to.

See Also

position(), hposition()

void Fl_Browser_::redraw_line (void ∗ *item*) `[protected]`

This method should be called when the contents of `item` has changed, but not its height.

Parameters

in	*item*	The item that needs to be redrawn.

See Also

redraw_lines(), redraw_line()

void Fl_Browser_::redraw_lines ( ) `[inline]`, `[protected]`

This method will cause the entire list to be redrawn.

See Also

redraw_lines(), redraw_line()

void Fl_Browser_::replacing (void ∗ *a*, void ∗ *b*) `[protected]`

This method should be used when item `a` is being replaced by item `b`.

It allows the Fl_Browser_ to update its cache data as needed, schedules a redraw for the item being changed, and tries to maintain the selection. This method does not actually replace the item, but handles the follow up bookkeeping after the item has just been replaced.

Parameters

in	*a*	Item being replaced
in	*b*	Item to replace 'a'

void Fl_Browser_::resize (int *X*, int *Y*, int *W*, int *H*) `[virtual]`

Repositions and/or resizes the browser.

Parameters

in	*X,Y,W,H*	The new position and size for the browser, in pixels.

Reimplemented from Fl_Widget.

void Fl_Browser_::scrollbar_left ( ) `[inline]`

Moves the vertical scrollbar to the lefthand side of the list.

For back compatibility.

void Fl_Browser_::scrollbar_right ( ) `[inline]`

Moves the vertical scrollbar to the righthand side of the list.

For back compatibility.

int Fl_Browser_::scrollbar_size ( ) const `[inline]`

Gets the current size of the scrollbars' troughs, in pixels.

If this value is zero (default), this widget will use the Fl::scrollbar_size() value as the scrollbar's width.

Returns

Scrollbar size in pixels, or 0 if the global Fl::scrollbar_size() is being used.

See Also

Fl::scrollbar_size(int)

void Fl_Browser_::scrollbar_size (int *newSize*) `[inline]`

Sets the pixel size of the scrollbars' troughs to `newSize`, in pixels.

Normally you should not need this method, and should use Fl::scrollbar_size(int) instead to manage the size of ALL your widgets' scrollbars. This ensures your application has a consistent UI, is the default behavior, and is normally what you want.

Only use THIS method if you really need to override the global scrollbar size. The need for this should be rare.

Setting `newSize` to the special value of 0 causes the widget to track the global Fl::scrollbar_size(), which is the default.

Parameters

`in`	*newSize*	Sets the scrollbar size in pixels. If 0 (default), scrollbar size tracks the global Fl::scrollbar_size()

See Also

Fl::scrollbar_size()

int Fl_Browser_::scrollbar_width ( ) const `[inline]`

This method has been deprecated, existing for backwards compatibility only.

Use scrollbar_size() instead. This method always returns the global value Fl::scrollbar_size().

Returns

Always returns the global value Fl::scrollbar_size().

Todo This method should eventually be removed in 1.4+

void Fl_Browser_::scrollbar_width (int *width*) `[inline]`

This method has been deprecated, existing for backwards compatibility only.

Use scrollbar_size(int) instead. This method sets the global Fl::scrollbar_size(), and forces this instance of the widget to use it.

Todo This method should eventually be removed in 1.4+

int Fl_Browser_::select (void ∗ *item*, int *val* = `1`, int *docallbacks* = `0`)

Sets the selection state of `item` to `val`, and returns 1 if the state changed or 0 if it did not.

If `docallbacks` is non-zero, select tries to call the callback function for the widget.

Parameters

`in`	*item*	The item whose selection state is to be changed
`in`	*val*	The new selection state (1=select, 0=de-select)
`in`	*docallbacks*	If 1, invokes widget callback if item changed. If 0, doesn't do callback (default).

Returns

1 if state was changed, 0 if not.

int Fl_Browser_::select_only (void ∗ *item*, int *docallbacks* = `0`)

Selects `item` and returns 1 if the state changed or 0 if it did not.

Any other items in the list are deselected.

Parameters

in	*item*	The `item` to select.
in	*docallbacks*	If 1, invokes widget callback if item changed. If 0, doesn't do callback (default).

void∗ Fl_Browser_::selection ( ) const `[inline]`, `[protected]`

Returns the item currently selected, or NULL if there is no selection.

For multiple selection browsers this call returns the currently focused item, even if it is not selected. To find all selected items, call Fl_Multi_Browser::selected() for every item in question.

void Fl_Browser_::sort (int *flags* = `0`)

Sort the items in the browser based on `flags`.

item_swap(void∗, void∗) and item_text(void∗) must be implemented for this call.

Parameters

in	*flags*	FL_SORT_ASCENDING – sort in ascending order FL_SORT_DESCENDING – sort in descending order Values other than the above will cause undefined behavior Other flags may appear in the future.

Todo Add a flag to ignore case

void Fl_Browser_::swapping (void ∗ *a*, void ∗ *b*) `[protected]`

This method should be used when two items `a` and `b` are being swapped.

It allows the Fl_Browser_ to update its cache data as needed, schedules a redraw for the two items, and tries to maintain the current selection. This method does not actually swap items, but handles the follow up bookkeeping after items have been swapped.

Parameters

in	*a,b*	Items being swapped.

Fl_Font Fl_Browser_::textfont ( ) const `[inline]`

Gets the default text font for the lines in the browser.

See Also

textfont(), textsize(), textcolor()

31.8.5 Member Data Documentation

Fl_Scrollbar Fl_Browser_::hscrollbar

Horizontal scrollbar.

Public, so that it can be accessed directly.

Fl_Scrollbar Fl_Browser_::scrollbar

Vertical scrollbar.

Public, so that it can be accessed directly.

The documentation for this class was generated from the following files:

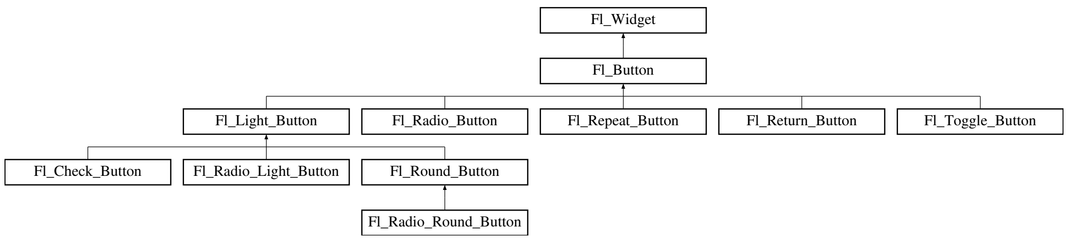
- Fl_Browser_.H
- Fl_Browser_.cxx

31.9 Fl_Button Class Reference

Buttons generate callbacks when they are clicked by the user.

`#include <Fl_Button.H>`

Inheritance diagram for Fl_Button:



Public Member Functions

- int clear ()

  Same as `value(0)`.
- Fl_Boxtype down_box () const

  Returns the current down box type, which is drawn when value() is non-zero.
- void down_box (Fl_Boxtype b)

  Sets the down box type.
- Fl_Color down_color () const

  (for backwards compatibility)
- void down_color (unsigned c)

  (for backwards compatibility)
- Fl_Button (int X, int Y, int W, int H, const char ∗L=0)

  The constructor creates the button using the given position, size, and label.
- virtual int handle (int)

  Handles the specified event.
- int set ()

  Same as `value(1)`.
- void setonly ()

  Turns on this button and turns off all other radio buttons in the group (calling `value(1)` *or* `set()` *does not do this).*
- int shortcut () const

  Returns the current shortcut key for the button.
- void shortcut (int s)

  Sets the shortcut key to `s`.
- void shortcut (const char ∗s)

  (for backwards compatibility)
- int value (int v)

  Sets the current value of the button.
- char value () const

  Returns the current value of the button (0 or 1).

Protected Member Functions

- virtual void draw ()

  Draws the widget.
- void **simulate_key_action** ()

Static Protected Member Functions

- static void **key_release_timeout** (void ∗)

Static Protected Attributes

- static Fl_Widget_Tracker ∗ **key_release_tracker** = 0

Additional Inherited Members

31.9.1 Detailed Description

Buttons generate callbacks when they are clicked by the user.

You control exactly when and how by changing the values for type() and when(). Buttons can also generate callbacks in response to `FL_SHORTCUT` events. The button can either have an explicit shortcut(int s) value or a letter shortcut can be indicated in the label() with an '&' character before it. For the label shortcut it does not matter if *Alt* is held down, but if you have an input field in the same window, the user will have to hold down the *Alt* key so that the input field does not eat the event first as an `FL_KEYBOARD` event.

Todo Refactor the doxygen comments for Fl_Button type() documentation.

For an Fl_Button object, the type() call returns one of:

- `FL_NORMAL_BUTTON` (0): value() remains unchanged after button press.
- `FL_TOGGLE_BUTTON:` value() is inverted after button press.
- `FL_RADIO_BUTTON:` value() is set to 1 after button press, and all other buttons in the current group with `type() == FL_RADIO_BUTTON` are set to zero.

Todo Refactor the doxygen comments for Fl_Button when() documentation.

For an Fl_Button object, the following when() values are useful, the default being `FL_WHEN_RELEASE:`

- `0`: The callback is not done, instead changed() is turned on.
- `FL_WHEN_RELEASE:` The callback is done after the user successfully clicks the button, or when a shortcut is typed.
- `FL_WHEN_CHANGED:` The callback is done each time the value() changes (when the user pushes and releases the button, and as the mouse is dragged around in and out of the button).

31.9.2 Constructor & Destructor Documentation

Fl_Button::Fl_Button (int *X*, int *Y*, int *W*, int *H*, const char ∗ *L* = `0`)

The constructor creates the button using the given position, size, and label.

The default box type is box(FL_UP_BOX).

You can control how the button is drawn when ON by setting down_box(). The default is FL_NO_BOX (0) which will select an appropriate box type using the normal (OFF) box type by using fl_down(box()).

Derived classes may handle this differently.

Parameters

in	*X,Y,W,H*	position and size of the widget
in	*L*	widget label, default is no label

31.9.3 Member Function Documentation

int Fl_Button::clear ( ) [inline]

Same as `value(0)`.

See Also

value(int v)

Fl_Boxtype Fl_Button::down_box ( ) const [inline]

Returns the current down box type, which is drawn when value() is non-zero.
Return values

Fl_Boxtype	

void Fl_Button::down_box (Fl_Boxtype *b*) [inline]

Sets the down box type.
The default value of 0 causes FLTK to figure out the correct matching down version of box().
Some derived classes (e.g. Fl_Round_Button and Fl_Light_Button use down_box() for special purposes. See docs of these classes.
Parameters

in	*b*	down box type

void Fl_Button::draw ( ) [protected], [virtual]

Draws the widget.
Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call redraw() instead.
Override this function to draw your own widgets.
If you ever need to call another widget's draw method *from within your own draw() method*, e.g. for an embedded scrollbar, you can do it (because draw() is virtual) like this:

```
Fl_Widget *s = &scroll;                // scroll is an embedded Fl_Scrollbar
s->draw();                   // calls Fl_Scrollbar::draw()
```

Implements Fl_Widget.
Reimplemented in Fl_Light_Button, and Fl_Return_Button.

int Fl_Button::handle (int *event*) [virtual]

Handles the specified event.
You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.
When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.
Most of the time, you want to call the inherited handle() method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters

in	*event*	the kind of event received

Return values

0	if the event was not used or understood
1	if the event was used and can be deleted

See Also

Fl_Event

Reimplemented from Fl_Widget.
Reimplemented in Fl_Light_Button, Fl_Return_Button, and Fl_Repeat_Button.

int Fl_Button::set ( ) `[inline]`

Same as `value(1)`.

See Also

value(int v)

int Fl_Button::shortcut ( ) const `[inline]`

Returns the current shortcut key for the button.
Return values

int	

void Fl_Button::shortcut (int *s*) `[inline]`

Sets the shortcut key to `s`.

Setting this overrides the use of '&' in the label(). The value is a bitwise OR of a key and a set of shift flags, for example: `FL_ALT | 'a'`, or `FL_ALT | (FL_F + 10)`, or just `'a'`. A value of 0 disables the shortcut.

The key can be any value returned by Fl::event_key(), but will usually be an ASCII letter. Use a lower-case letter unless you require the shift key to be held down.

The shift flags can be any set of values accepted by Fl::event_state(). If the bit is on, that shift key must be pushed. Meta, Alt, Ctrl, and Shift must be off if they are not in the shift flags (zero for the other bits indicates a "don't care" setting).
Parameters

in	*s*	bitwise OR of key and shift flags

int Fl_Button::value (int *v*)

Sets the current value of the button.

A non-zero value sets the button to 1 (ON), and zero sets it to 0 (OFF).
Parameters

in	*v*	button value.

See Also

set(), clear()

The documentation for this class was generated from the following files:

- Fl_Button.H
- Fl_Button.cxx

31.10 Fl_Cairo_State Class Reference

Contains all the necessary info on the current cairo context.

```
#include <Fl_Cairo.H>
```

Public Member Functions

- bool autolink () const

  Gets the autolink option. See Fl::cairo_autolink_context(bool)
- void autolink (bool b)

  Sets the autolink option, only available with –enable-cairoext.
- cairo_t ∗ cc () const

  Gets the current cairo context.
- void cc (cairo_t ∗c, bool own=true)

  Sets the current cairo context.
- void gc (void ∗c)

  Sets the gc `c` to keep track on.
- void ∗ gc () const

  Gets the last gc attached to a cc.
- void window (void ∗w)

  Sets the window `w` to keep track on.
- void ∗ window () const

  Gets the last window attached to a cc.

31.10.1 Detailed Description

Contains all the necessary info on the current cairo context.

A private internal & unique corresponding object is created to permit cairo context state handling while keeping it opaque. For internal use only.

Note

Only available when configure has the –enable-cairo option

31.10.2 Member Function Documentation

void Fl_Cairo_State::cc (cairo_t ∗ *c*, bool *own* = `true`) `[inline]`

Sets the current cairo context.

`own` == *true* (the default) indicates that the cairo context `c` will be deleted by FLTK internally when another cc is set later.

`own` == *false* indicates cc deletion is handled externally by the user program.

The documentation for this class was generated from the following files:

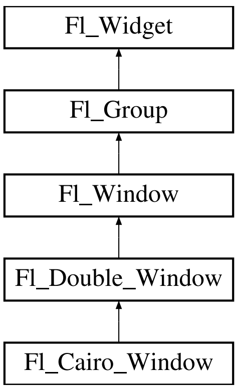
- Fl_Cairo.H
- Fl_Cairo.cxx

31.11 Fl_Cairo_Window Class Reference

This defines a pre-configured cairo fltk window.

`#include <Fl_Cairo_Window.H>`

Inheritance diagram for Fl_Cairo_Window:



Public Types

- typedef void(∗ cairo_draw_cb)(Fl_Cairo_Window ∗self, cairo_t ∗def)

  This defines the cairo draw callback prototype that you must further.

Public Member Functions

- **Fl_Cairo_Window** (int w, int h)
- void set_draw_cb (cairo_draw_cb cb)

  You must provide a draw callback which will implement your cairo rendering.

Protected Member Functions

- void draw ()

  Overloaded to provide cairo callback support.

Additional Inherited Members

31.11.1 Detailed Description

This defines a pre-configured cairo fltk window.

This class overloads the virtual draw() method for you, so that the only thing you have to do is to provide your cairo code. All cairo context handling is achieved transparently.

Note

> You can alternatively define your custom cairo fltk window, and thus at least override the draw() method to provide custom cairo support. In this case you will probably use Fl::cairo_make_current(-Fl_Window∗) to attach a context to your window. You should do it only when your window is the current window.

See Also

> Fl_Window::current()

31.11.2 Member Function Documentation

void Fl_Cairo_Window::set_draw_cb (cairo_draw_cb *cb*) `[inline]`

You must provide a draw callback which will implement your cairo rendering.

This method will permit you to set your cairo callback to `cb`.

The documentation for this class was generated from the following file:

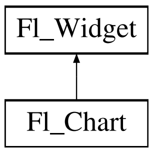
- Fl_Cairo_Window.H

31.12 Fl_Chart Class Reference

Fl_Chart displays simple charts.

`#include <Fl_Chart.H>`

Inheritance diagram for Fl_Chart:



Public Member Functions

- void add (double val, const char ∗str=0, unsigned col=0)

    Add the data value `val` with optional label `str` and color `col` to the chart.
- uchar autosize () const

    Get whether the chart will automatically adjust the bounds of the chart.
- void autosize (uchar n)

    Set whether the chart will automatically adjust the bounds of the chart.
- void bounds (double ∗a, double ∗b) const

    Gets the lower and upper bounds of the chart values.
- void bounds (double a, double b)

    Sets the lower and upper bounds of the chart values.
- void clear ()

    Removes all values from the chart.
- Fl_Chart (int X, int Y, int W, int H, const char ∗L=0)

    Create a new Fl_Chart widget using the given position, size and label string.
- void insert (int ind, double val, const char ∗str=0, unsigned col=0)

    Inserts a data value `val` at the given position `ind`.
- int maxsize () const

    Gets the maximum number of data values for a chart.
- void maxsize (int m)

    Set the maximum number of data values for a chart.
- void replace (int ind, double val, const char ∗str=0, unsigned col=0)

    Replace a data value `val` at the given position `ind`.
- int size () const

    Returns the number of data values in the chart.
- void **size** (int W, int H)
- Fl_Color textcolor () const

Gets the chart's text color.

- void textcolor (Fl_Color n)

  gets the chart's text color to `n`*.*

- Fl_Font textfont () const

  Gets the chart's text font.

- void textfont (Fl_Font s)

  Sets the chart's text font to `s`*.*

- Fl_Fontsize textsize () const

  Gets the chart's text size.

- void textsize (Fl_Fontsize s)

  gets the chart's text size to `s`*.*

- ~Fl_Chart ()

  Destroys the Fl_Chart widget and all of its data.

Protected Member Functions

- void draw ()

  Draws the widget.

Additional Inherited Members

31.12.1 Detailed Description

Fl_Chart displays simple charts.

It is provided for Forms compatibility.

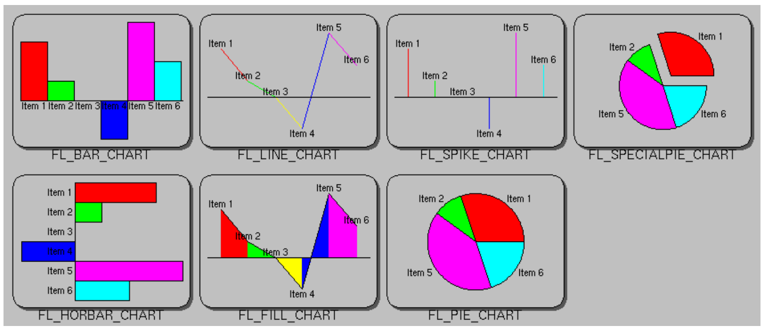


Figure 31.2: Fl_Chart

Todo Refactor Fl_Chart::type() information.

The type of an Fl_Chart object can be set using type(uchar t) to:

- `FL_BAR_CHART`: Each sample value is drawn as a vertical bar.
- `FL_FILLED_CHART`: The chart is filled from the bottom of the graph to the sample values.
- `FL_HORBAR_CHART`: Each sample value is drawn as a horizontal bar.
- `FL_LINE_CHART`: The chart is drawn as a polyline with vertices at each sample value.
- `FL_PIE_CHART`: A pie chart is drawn with each sample value being drawn as a proportionate slice in the circle.

- `FL_SPECIALPIE_CHART`: Like `FL_PIE_CHART`, but the first slice is separated from the pie.
- `FL_SPIKE_CHART`: Each sample value is drawn as a vertical line.

31.12.2 Constructor & Destructor Documentation

Fl_Chart::Fl_Chart (int *X*, int *Y*, int *W*, int *H*, const char ∗ *L* = `0`)

Create a new Fl_Chart widget using the given position, size and label string.
The default boxstyle is `FL_NO_BOX`.
Parameters

in	*X,Y,W,H*	position and size of the widget
in	*L*	widget label, default is no label

31.12.3 Member Function Documentation

void Fl_Chart::add (double *val*, const char ∗ *str* = `0`, unsigned *col* = `0`)

Add the data value `val` with optional label `str` and color `col` to the chart.
Parameters

in	*val*	data value
in	*str*	optional data label
in	*col*	optional data color

uchar Fl_Chart::autosize ( ) const `[inline]`

Get whether the chart will automatically adjust the bounds of the chart.

Returns

non-zero if auto-sizing is enabled and zero if disabled.

void Fl_Chart::autosize (uchar *n*) `[inline]`

Set whether the chart will automatically adjust the bounds of the chart.
Parameters

in	*n*	non-zero to enable automatic resizing, zero to disable.

void Fl_Chart::bounds (double ∗ *a*, double ∗ *b*) const `[inline]`

Gets the lower and upper bounds of the chart values.
Parameters

out	*a,b*	are set to lower, upper

void Fl_Chart::bounds (double *a*, double *b*)

Sets the lower and upper bounds of the chart values.

Parameters

in	*a,b*	are used to set lower, upper

void Fl_Chart::draw ( ) [protected], [virtual]

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call redraw() instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw() method*, e.g. for an embedded scrollbar, you can do it (because draw() is virtual) like this:

```
Fl_Widget *s = &scroll;              // scroll is an embedded Fl_Scrollbar
s->draw();                     // calls Fl_Scrollbar::draw()
```

Implements Fl_Widget.

void Fl_Chart::insert (int *ind*, double *val*, const char ∗ *str* = 0, unsigned *col* = 0)

Inserts a data value val at the given position ind.

Position 1 is the first data value.

Parameters

in	*ind*	insertion position
in	*val*	data value
in	*str*	optional data label
in	*col*	optional data color

void Fl_Chart::maxsize (int *m*)

Set the maximum number of data values for a chart.

If you do not call this method then the chart will be allowed to grow to any size depending on available memory.

Parameters

in	*m*	maximum number of data values allowed.

void Fl_Chart::replace (int *ind*, double *val*, const char ∗ *str* = 0, unsigned *col* = 0)

Replace a data value val at the given position ind.

Position 1 is the first data value.

Parameters

in	*ind*	insertion position
in	*val*	data value
in	*str*	optional data label
in	*col*	optional data color

void Fl_Chart::textcolor (Fl_Color *n*) [inline]

gets the chart's text color to n.

void Fl_Chart::textfont (Fl_Font *s*) [inline]

Sets the chart's text font to s.

void Fl_Chart::textsize (Fl_Fontsize *s*) `[inline]`

gets the chart's text size to `s`.

The documentation for this class was generated from the following files:

- Fl_Chart.H
- Fl_Chart.cxx

31.13 FL_CHART_ENTRY Struct Reference

For internal use only.

`#include <Fl_Chart.H>`

Public Attributes

- unsigned col

  For internal use only.
- char str [FL_CHART_LABEL_MAX+1]

  For internal use only.
- float val

  For internal use only.

31.13.1 Detailed Description

For internal use only.

31.13.2 Member Data Documentation

unsigned FL_CHART_ENTRY::col

For internal use only.

char FL_CHART_ENTRY::str[FL_CHART_LABEL_MAX+1]

For internal use only.

float FL_CHART_ENTRY::val

For internal use only.

The documentation for this struct was generated from the following file:

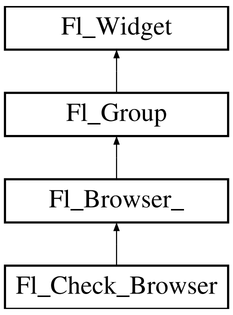
- Fl_Chart.H

31.14 Fl_Check_Browser Class Reference

The Fl_Check_Browser widget displays a scrolling list of text lines that may be selected and/or checked by the user.

`#include <Fl_Check_Browser.H>`

Inheritance diagram for Fl_Check_Browser:



Public Member Functions

- int add (char ∗s)

  Add a new unchecked line to the end of the browser.
- int add (char ∗s, int b)

  Add a new line to the end of the browser.
- int add (const char ∗s)

  See int Fl_Check_Browser::add(char ∗s)
- int add (const char ∗s, int b)

  See int Fl_Check_Browser::add(char ∗s)
- void check_all ()

  Sets all the items checked.
- void check_none ()

  Sets all the items unchecked.
- int checked (int item) const

  Gets the current status of item item.
- void checked (int item, int b)

  Sets the check status of item item to b.
- void clear ()

  Remove every item from the browser.
- Fl_Check_Browser (int x, int y, int w, int h, const char ∗l=0)

  The constructor makes an empty browser.
- int nchecked () const

  Returns how many items are currently checked.
- int nitems () const

  Returns how many lines are in the browser.
- int remove (int item)

  Remove line n and make the browser one line shorter.
- void set_checked (int item)

  Equivalent to Fl_Check_Browser::checked(item, 1).
- char ∗ text (int item) const

  Return a pointer to an internal buffer holding item item's text.
- int value () const

  Returns the index of the currently selected item.
- ∼Fl_Check_Browser ()

  The destructor deletes all list items and destroys the browser.

Protected Member Functions

- int handle (int)

  Handles the `event` within the normal widget bounding box.

Additional Inherited Members

31.14.1 Detailed Description

The Fl_Check_Browser widget displays a scrolling list of text lines that may be selected and/or checked by the user.

31.14.2 Constructor & Destructor Documentation

Fl_Check_Browser::Fl_Check_Browser (int *X*, int *Y*, int *W*, int *H*, const char * *l* = *0*)

The constructor makes an empty browser.

Fl_Check_Browser::~Fl_Check_Browser ( ) `[inline]`

The destructor deletes all list items and destroys the browser.

31.14.3 Member Function Documentation

int Fl_Check_Browser::add (char * *s*)

Add a new unchecked line to the end of the browser.

See Also

add(char *s, int b)

int Fl_Check_Browser::add (char * *s*, int *b*)

Add a new line to the end of the browser.

The text is copied using the strdup() function. It may also be NULL to make a blank line. It can set the item checked if `b` is not 0.

void Fl_Check_Browser::check_all ( )

Sets all the items checked.

void Fl_Check_Browser::check_none ( )

Sets all the items unchecked.

int Fl_Check_Browser::checked (int *i*) const

Gets the current status of item item.

void Fl_Check_Browser::checked (int *i*, int *b*)

Sets the check status of item item to b.

void Fl_Check_Browser::clear ( )

Remove every item from the browser.

int Fl_Check_Browser::handle (int *event*) `[protected]`, `[virtual]`

Handles the `event` within the normal widget bounding box.

Parameters

in	*event*	The event to process.

Returns

1 if event was processed, 0 if not.

Reimplemented from Fl_Browser_.

int Fl_Check_Browser::nchecked ( ) const [inline]

Returns how many items are currently checked.

int Fl_Check_Browser::nitems ( ) const [inline]

Returns how many lines are in the browser.
The last line number is equal to this.

int Fl_Check_Browser::remove (int *item*)

Remove line n and make the browser one line shorter.
Returns the number of lines left in the browser.

void Fl_Check_Browser::set_checked (int *item*) [inline]

Equivalent to Fl_Check_Browser::checked(item, 1).

char ∗ Fl_Check_Browser::text (int *i*) const

Return a pointer to an internal buffer holding item item's text.

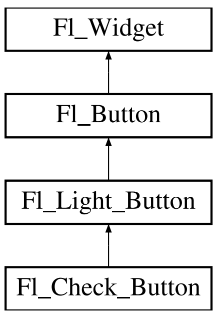
int Fl_Check_Browser::value ( ) const

Returns the index of the currently selected item.
The documentation for this class was generated from the following files:

- Fl_Check_Browser.H
- Fl_Check_Browser.cxx

31.15 Fl_Check_Button Class Reference

A button with a ”checkmark” to show its status.
Inheritance diagram for Fl_Check_Button:



Public Member Functions

- Fl_Check_Button (int X, int Y, int W, int H, const char ∗L=0)

  Creates a new Fl_Check_Button widget using the given position, size, and label string.

Additional Inherited Members

31.15.1 Detailed Description

A button with a "checkmark" to show its status.

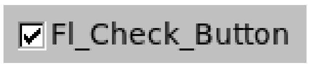


Figure 31.3: Fl_Check_Button

Buttons generate callbacks when they are clicked by the user. You control exactly when and how by changing the values for type() and when().

The Fl_Check_Button subclass displays its "ON" state by showing a "checkmark" rather than drawing itself pushed in.

31.15.2 Constructor & Destructor Documentation

Fl_Check_Button::Fl_Check_Button (int *X*, int *Y*, int *W*, int *H*, const char ∗ *L* = `0`)

Creates a new Fl_Check_Button widget using the given position, size, and label string.

The default box type is FL_NO_BOX, which draws the label w/o a box right of the checkmark.

The selection_color() sets the color of the checkmark. Default is FL_FOREGROUND_COLOR (usually black).

You can use down_box() to change the box type of the checkmark. Default is FL_DOWN_BOX.

Parameters

`in`	*X,Y,W,H*	position and size of the widget
`in`	*L*	widget label, default is no label

The documentation for this class was generated from the following files:

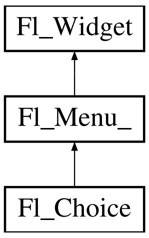
- Fl_Check_Button.H
- Fl_Check_Button.cxx

31.16 Fl_Choice Class Reference

A button that is used to pop up a menu.

`#include <Fl_Choice.H>`

Inheritance diagram for Fl_Choice:



Public Member Functions

- Fl_Choice (int X, int Y, int W, int H, const char ∗L=0)

  Create a new Fl_Choice widget using the given position, size and label string.

- int handle (int)

  Handles the specified event.

- int value () const

  Gets the index of the last item chosen by the user.

- int value (int v)

  Sets the currently selected value using the index into the menu item array.

- int value (const Fl_Menu_Item ∗v)

  Sets the currently selected value using a pointer to menu item.

Protected Member Functions

- void draw ()

  Draws the widget.

Additional Inherited Members

31.16.1 Detailed Description

A button that is used to pop up a menu.

This is a button that, when pushed, pops up a menu (or hierarchy of menus) defined by an array of Fl_Menu_Item objects. Motif calls this an OptionButton.

The only difference between this and a Fl_Menu_Button is that the name of the most recent chosen menu item is displayed inside the box, while the label is displayed outside the box. However, since the use of this is most often to control a single variable rather than do individual callbacks, some of the Fl_Menu_Button methods are redescribed here in those terms.

When the user clicks a menu item, value() is set to that item and then:

```
- The item's callback is done if one has been set; the
  Fl_Choice is passed as the Fl_Widget* argument,
  along with any userdata configured for the callback.

- If the item does not have a callback, the Fl_Choice widget's
  callback is done instead, along with any userdata configured
  for it.  The callback can determine which item was picked using
  value(), mvalue(), item_pathname(), etc.
```

All three mouse buttons pop up the menu. The Forms behavior of the first two buttons to increment/decrement the choice is not implemented. This could be added with a subclass, however.

The menu will also pop up in response to shortcuts indicated by putting a '&' character in the label(). See Fl_Button::shortcut(int s) for a description of this.

Typing the shortcut() of any of the items will do exactly the same as when you pick the item with the mouse. The '&' character in item names are only looked at when the menu is popped up, however.

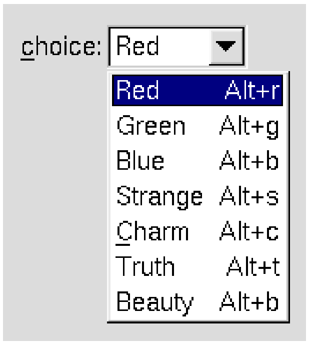


Figure 31.4: Fl_Choice

Todo Refactor the doxygen comments for Fl_Choice changed() documentation.

- `int Fl_Widget::changed() const` This value is true the user picks a different value. *It is turned off by value() and just before doing a callback (the callback can turn it back on if desired).*
- `void Fl_Widget::set_changed()` This method sets the changed() flag.
- `void Fl_Widget::clear_changed()` This method clears the changed() flag.
- `Fl_Boxtype Fl_Choice::down_box() const` Gets the current down box, which is used when the menu is popped up. The default down box type is `FL_DOWN_BOX`.
- `void Fl_Choice::down_box(Fl_Boxtype b)` Sets the current down box type to `b`.

31.16.2 Constructor & Destructor Documentation

Fl_Choice::Fl_Choice (int *X*, int *Y*, int *W*, int *H*, const char ∗ *L* = `0`)

Create a new Fl_Choice widget using the given position, size and label string.

The default boxtype is `FL_UP_BOX`.

The constructor sets menu() to NULL. See Fl_Menu_ for the methods to set or change the menu.

Parameters

in	*X,Y,W,H*	position and size of the widget
in	*L*	widget label, default is no label

31.16.3 Member Function Documentation

void Fl_Choice::draw ( ) `[protected]`, `[virtual]`

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call redraw() instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw() method*, e.g. for an embedded scrollbar, you can do it (because draw() is virtual) like this:

```
Fl_Widget *s = &scroll;                // scroll is an embedded Fl_Scrollbar
s->draw();                        // calls Fl_Scrollbar::draw()
```

Implements Fl_Widget.

int Fl_Choice::handle (int *event*) `[virtual]`

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited handle() method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters

`in`	*event*	the kind of event received

Return values

0	if the event was not used or understood
1	if the event was used and can be deleted

See Also

Fl_Event

Reimplemented from Fl_Widget.

int Fl_Choice::value ( ) const `[inline]`

Gets the index of the last item chosen by the user.

The index is zero initially.

int Fl_Choice::value (int *v*)

Sets the currently selected value using the index into the menu item array.

Changing the selected value causes a redraw().

Parameters

`in`	*v*	index of value in the menu item array.

Returns

non-zero if the new value is different to the old one.

int Fl_Choice::value (const Fl_Menu_Item $*$ *v*)

Sets the currently selected value using a pointer to menu item.

Changing the selected value causes a redraw().

Parameters

`in`	*v*	pointer to menu item in the menu item array.

Returns

non-zero if the new value is different to the old one.

The documentation for this class was generated from the following files:

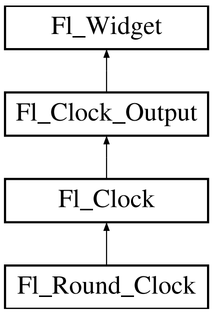
- Fl_Choice.H
- Fl_Choice.cxx

31.17 Fl_Clock Class Reference

This widget provides a round analog clock display.

```
#include <Fl_Clock.H>
```

Inheritance diagram for Fl_Clock:



Public Member Functions

- Fl_Clock (int X, int Y, int W, int H, const char ∗L=0)

  Create an Fl_Clock widget using the given position, size, and label string.
- Fl_Clock (uchar t, int X, int Y, int W, int H, const char ∗L)

  Create an Fl_Clock widget using the given boxtype, position, size, and label string.
- int handle (int)

  Handles the specified event.
- ∼Fl_Clock ()

  The destructor removes the clock.

Additional Inherited Members

31.17.1 Detailed Description

This widget provides a round analog clock display.

Fl_Clock is provided for Forms compatibility. It installs a 1-second timeout callback using Fl::add_-timeout(). You can choose the rounded or square type of the clock with type(), see below.

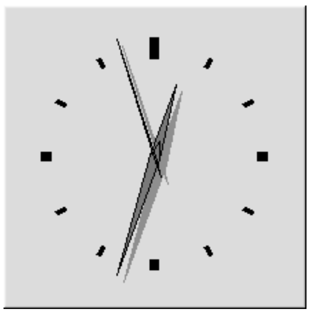


Figure 31.5: FL_SQUARE_CLOCK type

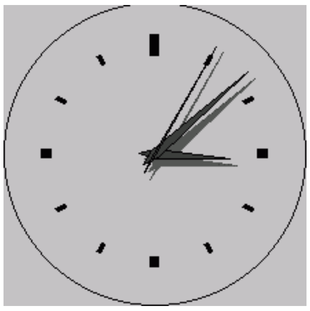


Figure 31.6: FL_ROUND_CLOCK type

31.17.2 Constructor & Destructor Documentation

Fl_Clock::Fl_Clock (int *X,* int *Y,* int *W,* int *H,* const char ∗ *L* = *0*)

Create an Fl_Clock widget using the given position, size, and label string.

The default boxtype is `FL_NO_BOX`.

Parameters

in	*X,Y,W,H*	position and size of the widget
in	*L*	widget label, default is no label

Fl_Clock::Fl_Clock (uchar *t,* int *X,* int *Y,* int *W,* int *H,* const char ∗ *L*)

Create an Fl_Clock widget using the given boxtype, position, size, and label string.

Parameters

in	*t*	boxtype
in	*X,Y,W,H*	position and size of the widget
in	*L*	widget label, default is no label

31.17.3 Member Function Documentation

int Fl_Clock::handle (int *event*) `[virtual]`

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited handle() method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters

in	*event*	the kind of event received

Return values

0	if the event was not used or understood
1	if the event was used and can be deleted

See Also

Fl_Event

Reimplemented from Fl_Widget.
The documentation for this class was generated from the following files:

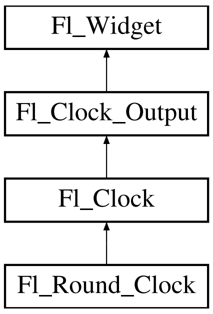
- Fl_Clock.H
- Fl_Clock.cxx

31.18 Fl_Clock_Output Class Reference

This widget can be used to display a program-supplied time.

```
#include <Fl_Clock.H>
```

Inheritance diagram for Fl_Clock_Output:



Public Member Functions

- Fl_Clock_Output (int X, int Y, int W, int H, const char *L=0)

  Create a new Fl_Clock_Output widget with the given position, size and label.
- int hour () const

  Returns the displayed hour (0 to 23).
- int minute () const

  Returns the displayed minute (0 to 59).
- int second () const

  Returns the displayed second (0 to 60, 60=leap second).
- void value (ulong v)

  Set the displayed time.
- void value (int H, int m, int s)

  Set the displayed time.
- ulong value () const

  Returns the displayed time.

Protected Member Functions

- void draw ()

  Draw clock with current position and size.
- void draw (int X, int Y, int W, int H)

  Draw clock with the given position and size.

Additional Inherited Members

31.18.1 Detailed Description

This widget can be used to display a program-supplied time.

The time shown on the clock is not updated. To display the current time, use Fl_Clock instead.

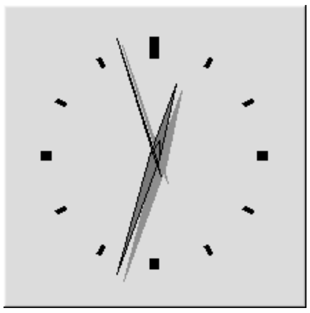


Figure 31.7: FL_SQUARE_CLOCK type

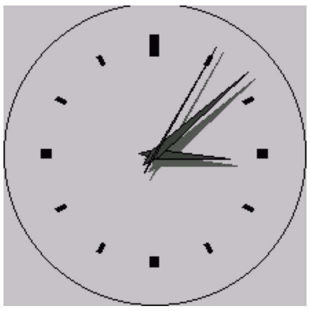


Figure 31.8: FL_ROUND_CLOCK type

31.18.2 Constructor & Destructor Documentation

Fl_Clock_Output::Fl_Clock_Output (int *X*, int *Y*, int *W*, int *H*, const char ∗ *L* = `0`)

Create a new Fl_Clock_Output widget with the given position, size and label.

The default boxtype is `FL_NO_BOX`.

Parameters

`in`	*X,Y,W,H*	position and size of the widget
`in`	*L*	widget label, default is no label

31.18.3 Member Function Documentation

void Fl_Clock_Output::draw (int *X*, int *Y*, int *W*, int *H*) `[protected]`

Draw clock with the given position and size.

Parameters

`in`	*X,Y,W,H*	position and size

int Fl_Clock_Output::hour ( ) const `[inline]`

Returns the displayed hour (0 to 23).

See Also

value(), minute(), second()

int Fl_Clock_Output::minute ( ) const `[inline]`

Returns the displayed minute (0 to 59).

See Also

value(), hour(), second()

int Fl_Clock_Output::second ( ) const `[inline]`

Returns the displayed second (0 to 60, 60=leap second).

See Also

value(), hour(), minute()

void Fl_Clock_Output::value (ulong *v*)

Set the displayed time.
Set the time in seconds since the UNIX epoch (January 1, 1970).
Parameters

`in`	*v*	seconds since epoch

See Also

value()

void Fl_Clock_Output::value (int *H*, int *m*, int *s*)

Set the displayed time.
Set the time in hours, minutes, and seconds.
Parameters

`in`	*H,m,s*	displayed time

See Also

hour(), minute(), second()

ulong Fl_Clock_Output::value ( ) const `[inline]`

Returns the displayed time.

Returns the time in seconds since the UNIX epoch (January 1, 1970).

See Also

value(ulong)

The documentation for this class was generated from the following files:

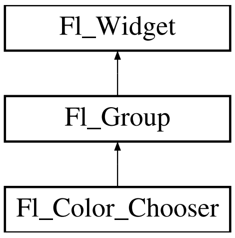
- Fl_Clock.H
- Fl_Clock.cxx

31.19 Fl_Color_Chooser Class Reference

The Fl_Color_Chooser widget provides a standard RGB color chooser.

`#include <Fl_Color_Chooser.H>`

Inheritance diagram for Fl_Color_Chooser:



Public Member Functions

- double b () const

  Returns the current blue value.
- Fl_Color_Chooser (int X, int Y, int W, int H, const char ∗L=0)

  Creates a new Fl_Color_Chooser widget using the given position, size, and label string.
- double g () const

  Returns the current green value.
- int hsv (double H, double S, double V)

  Set the hsv values.
- double hue () const

  Returns the current hue.
- int mode ()

  Returns which Fl_Color_Chooser variant is currently active.
- void mode (int newMode)

  Set which Fl_Color_Chooser variant is currently active.
- double r () const

  Returns the current red value.
- int rgb (double R, double G, double B)

  Sets the current rgb color values.
- double saturation () const

  Returns the saturation.
- double value () const

  Returns the value/brightness.

Static Public Member Functions

- static void hsv2rgb (double H, double S, double V, double &R, double &G, double &B)

  This static method converts HSV colors to RGB colorspace.
- static void rgb2hsv (double R, double G, double B, double &H, double &S, double &V)

  This static method converts RGB colors to HSV colorspace.

Related Functions

(Note that these are not member functions.)

- int fl_color_chooser (const char ∗name, double &r, double &g, double &b, int cmode)

  Pops up a window to let the user pick an arbitrary RGB color.
- int fl_color_chooser (const char ∗name, uchar &r, uchar &g, uchar &b, int cmode)

  Pops up a window to let the user pick an arbitrary RGB color.

Additional Inherited Members

31.19.1 Detailed Description

The Fl_Color_Chooser widget provides a standard RGB color chooser.

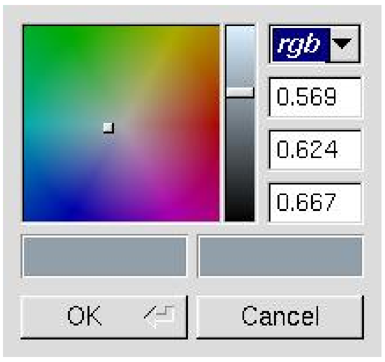


Figure 31.9: fl_color_chooser()

You can place any number of the widgets into a panel of your own design. The diagram shows the widget as part of a color chooser dialog created by the fl_color_chooser() function. The Fl_Color_Chooser widget contains the hue box, value slider, and rgb input fields from the above diagram (it does not have the color chips or the Cancel or OK buttons). The callback is done every time the user changes the rgb value. It is not done if they move the hue control in a way that produces the *same* rgb value, such as when saturation or value is zero.

The fl_color_chooser() function pops up a window to let the user pick an arbitrary RGB color. They can pick the hue and saturation in the "hue box" on the left (hold down CTRL to just change the saturation), and the brightness using the vertical slider. Or they can type the 8-bit numbers into the RGB Fl_Value_Input fields, or drag the mouse across them to adjust them. The pull-down menu lets the user set the input fields to show RGB, HSV, or 8-bit RGB (0 to 255).

fl_color_chooser() returns non-zero if the user picks ok, and updates the RGB values. If the user picks cancel or closes the window this returns zero and leaves RGB unchanged.

If you use the color chooser on an 8-bit screen, it will allocate all the available colors, leaving you no space to exactly represent the color the user picks! You can however use fl_rectf() to fill a region with a simulated color using dithering.

31.19.2 Constructor & Destructor Documentation

Fl_Color_Chooser::Fl_Color_Chooser (int *X*, int *Y*, int *W*, int *H*, const char ∗ *L* = `0`)

Creates a new Fl_Color_Chooser widget using the given position, size, and label string.

The recommended dimensions are 200x95. The color is initialized to black.

Parameters

`in`	*X,Y,W,H*	position and size of the widget
`in`	*L*	widget label, default is no label

31.19.3 Member Function Documentation

double Fl_Color_Chooser::b ( ) const `[inline]`

Returns the current blue value.

0 <= b <= 1.

double Fl_Color_Chooser::g ( ) const `[inline]`

Returns the current green value.

0 <= g <= 1.

int Fl_Color_Chooser::hsv (double *H*, double *S*, double *V*)

Set the hsv values.

The passed values are clamped (or for hue, modulus 6 is used) to get legal values. Does not do the callback.

Parameters

`in`	*H,S,V*	color components.

Returns

1 if a new hsv value was set, 0 if the hsv value was the previous one.

void Fl_Color_Chooser::hsv2rgb (double *H*, double *S*, double *V*, double & *R*, double & *G*, double & *B*) `[static]`

This *static* method converts HSV colors to RGB colorspace.

Parameters

`in`	*H,S,V*	color components
`out`	*R,G,B*	color components

double Fl_Color_Chooser::hue ( ) const `[inline]`

Returns the current hue.

0 <= hue < 6. Zero is red, one is yellow, two is green, etc. *This value is convenient for the internal calculations - some other systems consider hue to run from zero to one, or from 0 to 360.*

int Fl_Color_Chooser::mode ( ) `[inline]`

Returns which Fl_Color_Chooser variant is currently active.

Returns

color modes are rgb(0), byte(1), hex(2), or hsv(3)

void Fl_Color_Chooser::mode (int *newMode*)

Set which Fl_Color_Chooser variant is currently active.
Parameters

`in`	*newMode*	color modes are rgb(0), byte(1), hex(2), or hsv(3)

double Fl_Color_Chooser::r ( ) const `[inline]`

Returns the current red value.
0 <= r <= 1.

int Fl_Color_Chooser::rgb (double *R*, double *G*, double *B*)

Sets the current rgb color values.
Does not do the callback. Does not clamp (but out of range values will produce psychedelic effects in the hue selector).
Parameters

`in`	*R,G,B*	color components.

Returns

1 if a new rgb value was set, 0 if the rgb value was the previous one.

void Fl_Color_Chooser::rgb2hsv (double *R*, double *G*, double *B*, double & *H*, double & *S*, double & *V*) `[static]`

This *static* method converts RGB colors to HSV colorspace.
Parameters

`in`	*R,G,B*	color components
`out`	*H,S,V*	color components

double Fl_Color_Chooser::saturation ( ) const `[inline]`

Returns the saturation.
0 <= saturation <= 1.

double Fl_Color_Chooser::value ( ) const `[inline]`

Returns the value/brightness.
0 <= value <= 1.
The documentation for this class was generated from the following files:

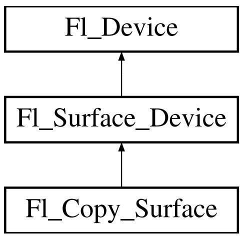
- Fl_Color_Chooser.H
- Fl_Color_Chooser.cxx

31.20 Fl_Copy_Surface Class Reference

Supports copying of graphical data to the clipboard.

`#include <Fl_Copy_Surface.H>`

Inheritance diagram for Fl_Copy_Surface:



Public Member Functions

- const char ∗ class_name ()

  Returns the name of the class of this object.
- void draw (Fl_Widget ∗widget, int delta_x=0, int delta_y=0)

  Copies a widget in the clipboard.
- void draw_decorated_window (Fl_Window ∗win, int delta_x=0, int delta_y=0)

  Copies a window and its borders and title bar to the clipboard.
- Fl_Copy_Surface (int w, int h)

  Constructor.
- int h ()

  Returns the pixel height of the copy surface.
- void set_current ()

  Make this surface the current drawing surface.
- int w ()

  Returns the pixel width of the copy surface.
- ∼Fl_Copy_Surface ()

  Destructor.

Static Public Attributes

- static const char ∗ **class_id** = ”Fl_Copy_Surface”

Additional Inherited Members

31.20.1 Detailed Description

Supports copying of graphical data to the clipboard.

After creation of an Fl_Copy_Surface object, call set_current() on it, and all subsequent graphics requests will be recorded in the clipboard. It's possible to draw widgets (using Fl_Copy_Surface::draw()) or to use any of the Drawing functions or the Color & Font functions. Finally, delete the Fl_Copy_Surface object to load the clipboard with the graphical data.

Fl_GL_Window 's can be copied to the clipboard as well.

Usage example:

```
Fl_Widget *g = ...; // a widget you want to copy to the clipboard
Fl_Copy_Surface *copy_surf = new Fl_Copy_Surface(g->
      w(), g->h()); // create an Fl_Copy_Surface object
copy_surf->set_current(); // direct graphics requests to the clipboard
fl_color(FL_WHITE); fl_rectf(0, 0, g->w(), g->h()); // draw a white background
copy_surf->draw(g); // draw the g widget in the clipboard
delete copy_surf; // after this, the clipboard is loaded
Fl_Display_Device::display_device()->
      set_current();  // direct graphics requests back to the display
```

Platform details:

- MSWindows: Transparent RGB images copy without transparency. The graphical data are copied to the clipboard as an 'enhanced metafile'.
- Mac OS: The graphical data are copied to the clipboard (a.k.a. pasteboard) in two 'flavors': 1) in vectorial form as PDF data; 2) in bitmap form as a TIFF image. Applications to which the clipboard content is pasted can use the flavor that suits them best.
- X11: the graphical data are copied to the clipboard as an image in BMP format.

31.20.2 Constructor & Destructor Documentation

Fl_Copy_Surface::Fl_Copy_Surface (int *w*, int *h*)

Constructor.
Parameters

w	and
h	are the width and height of the clipboard surface in pixels where drawing will occur.

31.20.3 Member Function Documentation

const char∗ Fl_Copy_Surface::class_name ( ) `[inline]`, `[virtual]`

Returns the name of the class of this object.

Use of the class_name() function is discouraged because it will be removed from future FLTK versions.

The class of an instance of an Fl_Device subclass can be checked with code such as:

```
if ( instance->class_name() == Fl_Printer::class_id ) { ... }
```

Reimplemented from Fl_Device.

void Fl_Copy_Surface::draw (Fl_Widget ∗ *widget*, int *delta_x* = `0`, int *delta_y* = `0`)

Copies a widget in the clipboard.
Parameters

widget	any FLTK widget (e.g., standard, custom, window, GL view) to copy
delta_x	and
delta_y	give the position in the clipboard of the top-left corner of the widget

void Fl_Copy_Surface::draw_decorated_window (Fl_Window ∗ *win*, int *delta_x* = `0`, int *delta_y* = `0`)

Copies a window and its borders and title bar to the clipboard.

Parameters

win	an FLTK window to copy
delta_x	and
delta_y	give the position in the clipboard of the top-left corner of the window's title bar

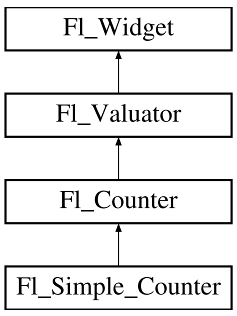
void Fl_Copy_Surface::set_current (void) [virtual]

Make this surface the current drawing surface.
This surface will receive all future graphics requests.
Reimplemented from Fl_Surface_Device.
The documentation for this class was generated from the following files:

- Fl_Copy_Surface.H
- Fl_Copy_Surface.cxx

31.21 Fl_Counter Class Reference

Controls a single floating point value with button (or keyboard) arrows.
`#include <Fl_Counter.H>`
Inheritance diagram for Fl_Counter:



Public Member Functions

- Fl_Counter (int X, int Y, int W, int H, const char *L=0)

  Creates a new Fl_Counter widget using the given position, size, and label string.
- int handle (int)

  Handles the specified event.
- void lstep (double a)

  Sets the increment for the large step buttons.
- void step (double a, double b)

  Sets the increments for the normal and large step buttons.
- void step (double a)

  Sets the increment for the normal step buttons.
- double step () const

  Returns the increment for normal step buttons.
- Fl_Color textcolor () const

  Gets the font color.
- void textcolor (Fl_Color s)

Sets the font color to `s`.

- Fl_Font textfont () const

  Gets the text font.
- void textfont (Fl_Font s)

  Sets the text font to `s`.
- Fl_Fontsize textsize () const

  Gets the font size.
- void textsize (Fl_Fontsize s)

  Sets the font size to `s`.
- ~Fl_Counter ()

  Destroys the valuator.

Protected Member Functions

- void draw ()

  Draws the widget.

Additional Inherited Members

31.21.1 Detailed Description

Controls a single floating point value with button (or keyboard) arrows.

Double arrows buttons achieve larger steps than simple arrows.

See Also

Fl_Spinner for value input with vertical step arrows.

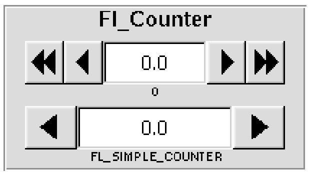


Figure 31.10: Fl_Counter

Todo Refactor the doxygen comments for Fl_Counter type() documentation.

The type of an Fl_Counter object can be set using type(uchar t) to:

- `FL_NORMAL_COUNTER`: Displays a counter with 4 arrow buttons.
- `FL_SIMPLE_COUNTER`: Displays a counter with only 2 arrow buttons.

31.21.2 Constructor & Destructor Documentation

Fl_Counter::Fl_Counter (int *X*, int *Y*, int *W*, int *H*, const char ∗ *L* = `0`)

Creates a new Fl_Counter widget using the given position, size, and label string.

The default type is FL_NORMAL_COUNTER.

Parameters

in	*X,Y,W,H*	position and size of the widget
in	*L*	widget label, default is no label

31.21.3 Member Function Documentation

void Fl_Counter::draw ( ) [protected], [virtual]

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call redraw() instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw() method*, e.g. for an embedded scrollbar, you can do it (because draw() is virtual) like this:

```
Fl_Widget *s = &scroll;              // scroll is an embedded Fl_Scrollbar
s->draw();                      // calls Fl_Scrollbar::draw()
```

Implements Fl_Widget.

int Fl_Counter::handle (int *event*) [virtual]

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited handle() method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters

in	*event*	the kind of event received

Return values

0	if the event was not used or understood
1	if the event was used and can be deleted

See Also

Fl_Event

Reimplemented from Fl_Widget.

void Fl_Counter::lstep (double *a*) [inline]

Sets the increment for the large step buttons.

The default value is 1.0.

Parameters

in	*a*	large step increment.

void Fl_Counter::step (double *a*, double *b*) [inline]

Sets the increments for the normal and large step buttons.

Parameters

in	*a,b*	normal and large step increments.

void Fl_Counter::step (double *a*) `[inline]`

Sets the increment for the normal step buttons.
Parameters

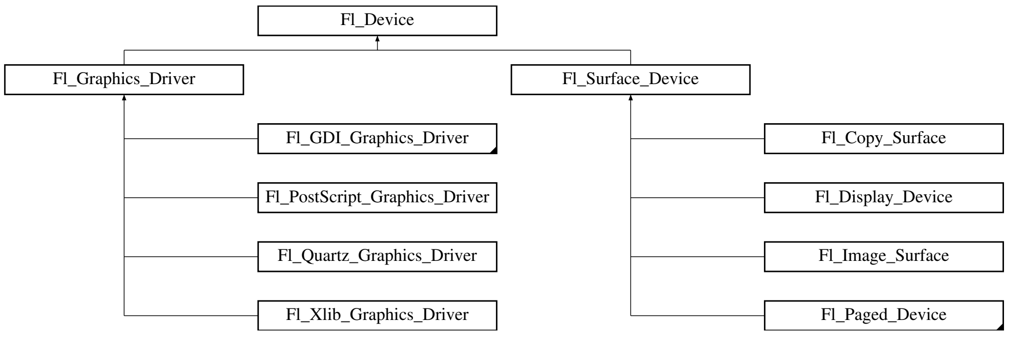
in	*a*	normal step increment.

The documentation for this class was generated from the following files:

- Fl_Counter.H
- Fl_Counter.cxx

31.22 Fl_Device Class Reference

All graphical output devices and all graphics systems.
`#include <Fl_Device.H>`
Inheritance diagram for Fl_Device:



Public Member Functions

- virtual const char ∗ class_name ()

  Returns the name of the class of this object.
- virtual ∼Fl_Device ()

  Virtual destructor.

Static Public Attributes

- static const char ∗ class_id = "Fl_Device"

  A string that identifies each subclass of Fl_Device.

31.22.1 Detailed Description

All graphical output devices and all graphics systems.
This class supports a rudimentary system of run-time type information.

31.22.2 Constructor & Destructor Documentation

virtual Fl_Device::∼Fl_Device ( ) `[inline]`, `[virtual]`

Virtual destructor.

The destructor of Fl_Device must be virtual to make the destructors of derived classes being called correctly on destruction.

31.22.3 Member Function Documentation

virtual const char∗ Fl_Device::class_name ( ) `[inline]`, `[virtual]`

Returns the name of the class of this object.

Use of the class_name() function is discouraged because it will be removed from future FLTK versions.

The class of an instance of an Fl_Device subclass can be checked with code such as:

```
if ( instance->class_name() == Fl_Printer::class_id ) { ... }
```

Reimplemented in Fl_Display_Device, Fl_Surface_Device, Fl_Xlib_Graphics_Driver, Fl_GDI_Printer_-Graphics_Driver, Fl_GDI_Graphics_Driver, Fl_Quartz_Graphics_Driver, Fl_Graphics_Driver, Fl_PostScript-_File_Device, Fl_Printer, Fl_Paged_Device, Fl_PostScript_Printer, Fl_Copy_Surface, Fl_System_Printer, Fl-_PostScript_Graphics_Driver, and Fl_Image_Surface.

31.22.4 Member Data Documentation

const char ∗ Fl_Device::class_id = "Fl_Device" `[static]`

A string that identifies each subclass of Fl_Device.

Function class_name() applied to a device of this class returns this string.

The documentation for this class was generated from the following files:

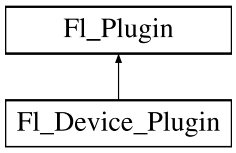
- Fl_Device.H
- Fl_Device.cxx

31.23 Fl_Device_Plugin Class Reference

This plugin socket allows the integration of new device drivers for special window or screen types.

`#include <Fl_Device.H>`

Inheritance diagram for Fl_Device_Plugin:



Public Member Functions

- Fl_Device_Plugin (const char ∗pluginName)

  The constructor.
- virtual const char ∗ klass ()

  Returns the class name.
- virtual const char ∗ name ()=0

  Returns the plugin name.

- virtual int print (Fl_Widget ∗w, int x, int y, int height)=0

  Prints a widget.
- Fl_RGB_Image ∗ rectangle_capture (Fl_Widget ∗widget, int x, int y, int w, int h)

  captures a rectangle of a widget as an image

31.23.1 Detailed Description

This plugin socket allows the integration of new device drivers for special window or screen types.

This class is not intended for use outside the FLTK library. It is currently used to provide an automated printing service and screen capture for OpenGL windows, if linked with fltk_gl.

31.23.2 Member Function Documentation

virtual int Fl_Device_Plugin::print (Fl_Widget ∗ *w,* int *x,* int *y,* int *height*) `[pure virtual]`

Prints a widget.

Parameters

w	the widget
x,y	offsets where to print relatively to coordinates origin
height	height of the current drawing area

Fl_RGB_Image∗ Fl_Device_Plugin::rectangle_capture (Fl_Widget ∗ *widget,* int *x,* int *y,* int *w,* int *h*) `[inline]`

captures a rectangle of a widget as an image

Returns

The captured pixels as an RGB image

The documentation for this class was generated from the following file:

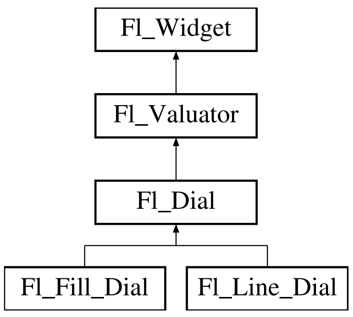
- Fl_Device.H

31.24 Fl_Dial Class Reference

The Fl_Dial widget provides a circular dial to control a single floating point value.

`#include <Fl_Dial.H>`

Inheritance diagram for Fl_Dial:



Public Member Functions

- short angle1 () const

  Sets Or gets the angles used for the minimum and maximum values.
- void angle1 (short a)

  See short angle1() const.
- short angle2 () const

  See short angle1() const.
- void angle2 (short a)

  See short angle1() const.
- void angles (short a, short b)

  See short angle1() const.
- Fl_Dial (int x, int y, int w, int h, const char ∗l=0)

  Creates a new Fl_Dial widget using the given position, size, and label string.
- int handle (int)

  Allow subclasses to handle event based on current position and size.

Protected Member Functions

- void draw (int X, int Y, int W, int H)

  Draws dial at given position and size.
- void draw ()

  Draws dial at current position and size.
- int handle (int event, int X, int Y, int W, int H)

  Allows subclasses to handle event based on given position and size.

Additional Inherited Members

31.24.1 Detailed Description

The Fl_Dial widget provides a circular dial to control a single floating point value.

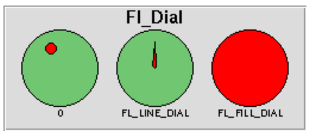


Figure 31.11: Fl_Dial

Use type() to set the type of the dial to:

- FL_NORMAL_DIAL - Draws a normal dial with a knob.
- FL_LINE_DIAL - Draws a dial with a line.
- FL_FILL_DIAL - Draws a dial with a filled arc.

31.24.2 Constructor & Destructor Documentation

Fl_Dial::Fl_Dial (int *X*, int *Y*, int *W*, int *H*, const char ∗ *l* = 0)

Creates a new Fl_Dial widget using the given position, size, and label string.

The default type is FL_NORMAL_DIAL.

31.24.3 Member Function Documentation

short Fl_Dial::angle1 ( ) const [inline]

Sets Or gets the angles used for the minimum and maximum values.

The default values are 45 and 315 (0 degrees is straight down and the angles progress clockwise). Normally angle1 is less than angle2, but if you reverse them the dial moves counter-clockwise.

void Fl_Dial::draw (int *X*, int *Y*, int *W*, int *H*) [protected]

Draws dial at given position and size.

Parameters

in	*X,Y,W,H*	position and size

int Fl_Dial::handle (int *event*, int *X*, int *Y*, int *W*, int *H*) [protected]

Allows subclasses to handle event based on given position and size.

Parameters

in	*event,X,Y,W,H*	event to handle, related position and size.

The documentation for this class was generated from the following files:

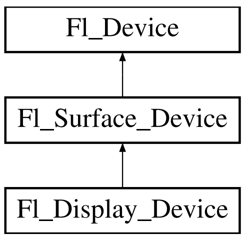
- Fl_Dial.H
- Fl_Dial.cxx

31.25 Fl_Display_Device Class Reference

A display to which the computer can draw.

`#include <Fl_Device.H>`

Inheritance diagram for Fl_Display_Device:



Public Member Functions

- const char ∗ class_name ()

  Returns the name of the class of this object.
- Fl_Display_Device (Fl_Graphics_Driver ∗graphics_driver)

  A constructor that sets the graphics driver used by the display.

Static Public Member Functions

- static Fl_Display_Device ∗ display_device ()

  Returns the platform display device.

Static Public Attributes

- static const char ∗ **class_id** = "Fl_Display_Device"

Additional Inherited Members

31.25.1 Detailed Description

A display to which the computer can draw.

When the program begins running, an Fl_Display_Device instance has been created and made the current drawing surface. There is no need to create any other object of this class.

31.25.2 Member Function Documentation

const char∗ Fl_Display_Device::class_name ( ) `[inline]`, `[virtual]`

Returns the name of the class of this object.

Use of the class_name() function is discouraged because it will be removed from future FLTK versions.

The class of an instance of an Fl_Device subclass can be checked with code such as:

```
if ( instance->class_name() == Fl_Printer::class_id ) { ... }
```

Reimplemented from Fl_Surface_Device.

Fl_Display_Device ∗ Fl_Display_Device::display_device ( ) `[static]`

Returns the platform display device.

The documentation for this class was generated from the following files:

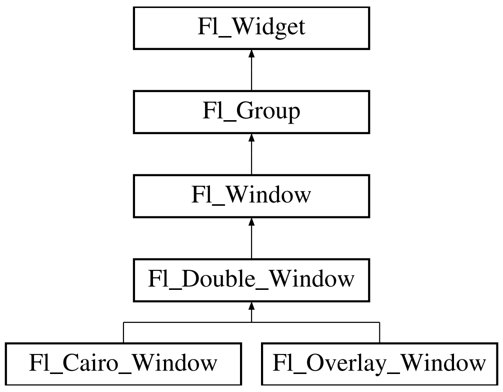
- Fl_Device.H
- Fl_Device.cxx

31.26 Fl_Double_Window Class Reference

The Fl_Double_Window provides a double-buffered window.

`#include <Fl_Double_Window.H>`

Inheritance diagram for Fl_Double_Window:



Public Member Functions

- Fl_Double_Window (int W, int H, const char ∗l=0)

  Creates a new Fl_Double_Window widget using the given position, size, and label (title) string.
- Fl_Double_Window (int X, int Y, int W, int H, const char ∗l=0)

  See Fl_Double_Window::Fl_Double_Window(int w, int h, const char ∗label = 0)
- void flush ()

  Forces the window to be redrawn.
- void hide ()

  Removes the window from the screen.
- void resize (int, int, int, int)

  Changes the size and position of the window.
- void show ()

  Puts the window on the screen.
- void **show** (int a, char ∗∗b)
- ~Fl_Double_Window ()

  The destructor also deletes all the children.

Protected Member Functions

- void flush (int eraseoverlay)

  Forces the window to be redrawn.

Protected Attributes

- char force_doublebuffering_

  Force double buffering, even if the OS already buffers windows (overlays need that on MacOS and Windows2000)

Additional Inherited Members

31.26.1 Detailed Description

The Fl_Double_Window provides a double-buffered window.

If possible this will use the X double buffering extension (Xdbe). If not, it will draw the window data into an off-screen pixmap, and then copy it to the on-screen window.

It is highly recommended that you put the following code before the first show() of *any* window in your program:

```
Fl::visual(FL_DOUBLE|FL_INDEX)
```

This makes sure you can use Xdbe on servers where double buffering does not exist for every visual.

31.26.2 Constructor & Destructor Documentation

Fl_Double_Window::~Fl_Double_Window ( )

The destructor *also deletes all the children*.

This allows a whole tree to be deleted at once, without having to keep a pointer to all the children in the user code.

31.26.3 Member Function Documentation

void Fl_Double_Window::flush (int *eraseoverlay*) `[protected]`

Forces the window to be redrawn.

Parameters

in	*eraseoverlay*	non-zero to erase overlay, zero to ignore

Fl_Overlay_Window relies on flush(1) copying the back buffer to the front everywhere, even if damage() == 0, thus erasing the overlay, and leaving the clip region set to the entire window.

void Fl_Double_Window::hide ( ) [virtual]

Removes the window from the screen.

If the window is already hidden or has not been shown then this does nothing and is harmless.

Reimplemented from Fl_Window.

Reimplemented in Fl_Overlay_Window.

void Fl_Double_Window::resize (int *X*, int *Y*, int *W*, int *H*) [virtual]

Changes the size and position of the window.

If shown() is true, these changes are communicated to the window server (which may refuse that size and cause a further resize). If shown() is false, the size and position are used when show() is called. See Fl_Group for the effect of resizing on the child widgets.

You can also call the Fl_Widget methods size(x,y) and position(w,h), which are inline wrappers for this virtual function.

A top-level window can not force, but merely suggest a position and size to the operating system. The window manager may not be willing or able to display a window at the desired position or with the given dimensions. It is up to the application developer to verify window parameters after the resize request.

Reimplemented from Fl_Window.

Reimplemented in Fl_Overlay_Window.

void Fl_Double_Window::show ( ) [virtual]

Puts the window on the screen.

Usually (on X) this has the side effect of opening the display.

If the window is already shown then it is restored and raised to the top. This is really convenient because your program can call show() at any time, even if the window is already up. It also means that show() serves the purpose of raise() in other toolkits.

Fl_Window::show(int argc, char **argv) is used for top-level windows and allows standard arguments to be parsed from the command-line.

Note

For some obscure reasons Fl_Window::show() resets the current group by calling Fl_Group::current(0). The comments in the code say "get rid of very common user bug: forgot end()". Although this is true it may have unwanted side effects if you show() an unrelated window (maybe for an error message or warning) while building a window or any other group widget.

Todo Check if we can remove resetting the current group in a later FLTK version (after 1.3.x). This may break "already broken" programs though if they rely on this "feature".

See Also

Fl_Window::show(int argc, char **argv)

Reimplemented from Fl_Window.

Reimplemented in Fl_Overlay_Window.

The documentation for this class was generated from the following files:

- Fl_Double_Window.H
- Fl_Double_Window.cxx

31.27 Fl_End Class Reference

This is a dummy class that allows you to end a Fl_Group in a constructor list of a class:

`#include <Fl_Group.H>`

Public Member Functions

- Fl_End ()

  All it does is calling Fl_Group::current()->end()

31.27.1 Detailed Description

This is a dummy class that allows you to end a Fl_Group in a constructor list of a class:

```
 class MyClass {
 Fl_Group group;
 Fl_Button button_in_group;
 Fl_End end;
 Fl_Button button_outside_group;
 MyClass();
};
MyClass::MyClass() :
 group(10,10,100,100),
 button_in_group(20,20,60,30),
 end(),
 button_outside_group(10,120,60,30)
{}
```

The documentation for this class was generated from the following file:

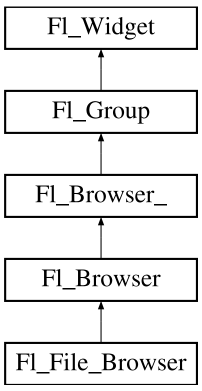
- Fl_Group.H

31.28 Fl_File_Browser Class Reference

The Fl_File_Browser widget displays a list of filenames, optionally with file-specific icons.

`#include <Fl_File_Browser.H>`

Inheritance diagram for Fl_File_Browser:



Public Types

- enum { **FILES**, **DIRECTORIES** }

Public Member Functions

- int filetype () const

  Sets or gets the file browser type, FILES or DIRECTORIES.
- void filetype (int t)

  Sets or gets the file browser type, FILES or DIRECTORIES.
- void filter (const char ∗pattern)

  Sets or gets the filename filter.
- const char ∗ filter () const

  Sets or gets the filename filter.
- Fl_File_Browser (int, int, int, int, const char ∗=0)

  The constructor creates the Fl_File_Browser widget at the specified position and size.
- uchar iconsize () const

  Sets or gets the size of the icons.
- void iconsize (uchar s)

  Sets or gets the size of the icons.
- int load (const char ∗directory, Fl_File_Sort_F ∗sort=fl_numericsort)

  Loads the specified directory into the browser.
- Fl_Fontsize **textsize** () const
- void **textsize** (Fl_Fontsize s)

Additional Inherited Members

31.28.1 Detailed Description

The Fl_File_Browser widget displays a list of filenames, optionally with file-specific icons.

31.28.2 Constructor & Destructor Documentation

Fl_File_Browser::Fl_File_Browser (int *X*, int *Y*, int *W*, int *H*, const char ∗ *l* = 0)

The constructor creates the Fl_File_Browser widget at the specified position and size.
The destructor destroys the widget and frees all memory that has been allocated.

31.28.3 Member Function Documentation

int Fl_File_Browser::filetype ( ) const [inline]

Sets or gets the file browser type, FILES or DIRECTORIES.
When set to FILES, both files and directories are shown. Otherwise only directories are shown.

void Fl_File_Browser::filetype (int *t*) [inline]

Sets or gets the file browser type, FILES or DIRECTORIES.
When set to FILES, both files and directories are shown. Otherwise only directories are shown.

void Fl_File_Browser::filter (const char ∗ *pattern*)

Sets or gets the filename filter.
The pattern matching uses the fl_filename_match() function in FLTK.

const char∗ Fl_File_Browser::filter ( ) const `[inline]`

Sets or gets the filename filter.
The pattern matching uses the fl_filename_match() function in FLTK.

uchar Fl_File_Browser::iconsize ( ) const `[inline]`

Sets or gets the size of the icons.
The default size is 20 pixels.

void Fl_File_Browser::iconsize (uchar *s*) `[inline]`

Sets or gets the size of the icons.
The default size is 20 pixels.

int Fl_File_Browser::load (const char ∗ *directory*, Fl_File_Sort_F ∗ *sort* = `fl_numericsort`)

Loads the specified directory into the browser.
If icons have been loaded then the correct icon is associated with each file in the list.
The sort argument specifies a sort function to be used with fl_filename_list().
The documentation for this class was generated from the following files:

- Fl_File_Browser.H
- Fl_File_Browser.cxx

31.29 Fl_File_Chooser Class Reference

The Fl_File_Chooser widget displays a standard file selection dialog that supports various selection modes.

Public Types

- enum { **SINGLE** = 0, **MULTI** = 1, **CREATE** = 2, **DIRECTORY** = 4 }

Public Member Functions

- Fl_Widget ∗ add_extra (Fl_Widget ∗gr)

  Adds extra widget at the bottom of Fl_File_Chooser window.
- void callback (void(∗cb)(Fl_File_Chooser ∗, void ∗), void ∗d=0)

  Sets the file chooser callback cb and associated data d.
- void color (Fl_Color c)

  Sets the background color of the Fl_File_Browser list.
- Fl_Color color ()

  Gets the background color of the Fl_File_Browser list.
- int count ()

  Returns the number of selected files.
- void directory (const char ∗d)

  Sets the current directory.
- char ∗ directory ()

  Gets the current directory.
- void filter (const char ∗p)

  Sets or gets the current filename filter patterns.
- const char ∗ filter ()

See void filter(const char ∗pattern)

- int filter_value ()

  Gets the current filename filter selection.
- void filter_value (int f)

  Sets the current filename filter selection.
- Fl_File_Chooser (const char ∗d, const char ∗p, int t, const char ∗title)

  The constructor creates the Fl_File_Chooser dialog shown.
- void hide ()

  Hides the Fl_File_Chooser window.
- void iconsize (uchar s)

  Sets the size of the icons in the Fl_File_Browser.
- uchar iconsize ()

  Gets the size of the icons in the Fl_File_Browser.
- void label (const char ∗l)

  Sets the title bar text for the Fl_File_Chooser.
- const char ∗ label ()

  Gets the title bar text for the Fl_File_Chooser.
- void ok_label (const char ∗l)

  Sets the label for the "ok" button in the Fl_File_Chooser.
- const char ∗ ok_label ()

  Gets the label for the "ok" button in the Fl_File_Chooser.
- void preview (int e)

  Enable or disable the preview tile.
- int preview () const

  Returns the current state of the preview box.
- void rescan ()

  Reloads the current directory in the Fl_File_Browser.
- void rescan_keep_filename ()

  Rescan the current directory without clearing the filename, then select the file if it is in the list.
- void show ()

  Shows the Fl_File_Chooser window.
- int shown ()

  Returns non-zero if the file chooser main window show() has been called (but not hide() see Fl_Window::shown()
- void textcolor (Fl_Color c)

  Sets the current Fl_File_Browser text color.
- Fl_Color textcolor ()

  Gets the current Fl_File_Browser text color.
- void textfont (Fl_Font f)

  Sets the current Fl_File_Browser text font.
- Fl_Font textfont ()

  Gets the current Fl_File_Browser text font.
- void textsize (Fl_Fontsize s)

  Sets the current Fl_File_Browser text size.
- Fl_Fontsize textsize ()

  Gets the current Fl_File_Browser text size.
- void type (int t)

Sets the current type of Fl_File_Chooser.

- int type ()

  Gets the current type of Fl_File_Chooser.
- void ∗ user_data () const

  Gets the file chooser user data.
- void user_data (void ∗d)

  Sets the file chooser user data d.
- const char ∗ value (int f=1)

  Gets the current value of the selected file(s).
- void value (const char ∗filename)

  Sets the current value of the selected file.
- int visible ()

  Returns 1 if the Fl_File_Chooser window is visible.
- ∼Fl_File_Chooser ()

  Destroys the widget and frees all memory used by it.

Public Attributes

- Fl_Button ∗ newButton

  The "new directory" button is exported so that application developers can control the appearance and use.
- Fl_Check_Button ∗ previewButton

  The "preview" button is exported so that application developers can control the appearance and use.
- Fl_Check_Button ∗ showHiddenButton

  When checked, hidden files (i.e., filename begins with dot) are displayed.

Static Public Attributes

- static const char ∗ add_favorites_label = "Add to Favorites"

  [standard text may be customized at run-time]
- static const char ∗ all_files_label = "All Files (∗)"

  [standard text may be customized at run-time]
- static const char ∗ custom_filter_label = "Custom Filter"

  [standard text may be customized at run-time]
- static const char ∗ existing_file_label = "Please choose an existing file!"

  [standard text may be customized at run-time]
- static const char ∗ favorites_label = "Favorites"

  [standard text may be customized at run-time]
- static const char ∗ filename_label = "Filename:"

  [standard text may be customized at run-time]
- static const char ∗ filesystems_label = "File Systems"

  [standard text may be customized at run-time]
- static const char ∗ hidden_label = "Show hidden files"

  [standard text may be customized at run-time]
- static const char ∗ manage_favorites_label = "Manage Favorites"

  [standard text may be customized at run-time]
- static const char ∗ new_directory_label = "New Directory?"

  [standard text may be customized at run-time]
- static const char ∗ new_directory_tooltip = "Create a new directory."

[standard text may be customized at run-time]

- static const char ∗ preview_label = ”Preview”

  [standard text may be customized at run-time]
- static const char ∗ save_label = ”Save”

  [standard text may be customized at run-time]
- static const char ∗ show_label = ”Show:”

  [standard text may be customized at run-time]
- static Fl_File_Sort_F ∗ sort = fl_numericsort

  the sort function that is used when loading the contents of a directory.

Related Functions

(Note that these are not member functions.)

- char ∗ fl_dir_chooser (const char ∗message, const char ∗fname, int relative)

  Shows a file chooser dialog and gets a directory.
- char ∗ fl_file_chooser (const char ∗message, const char ∗pat, const char ∗fname, int relative)

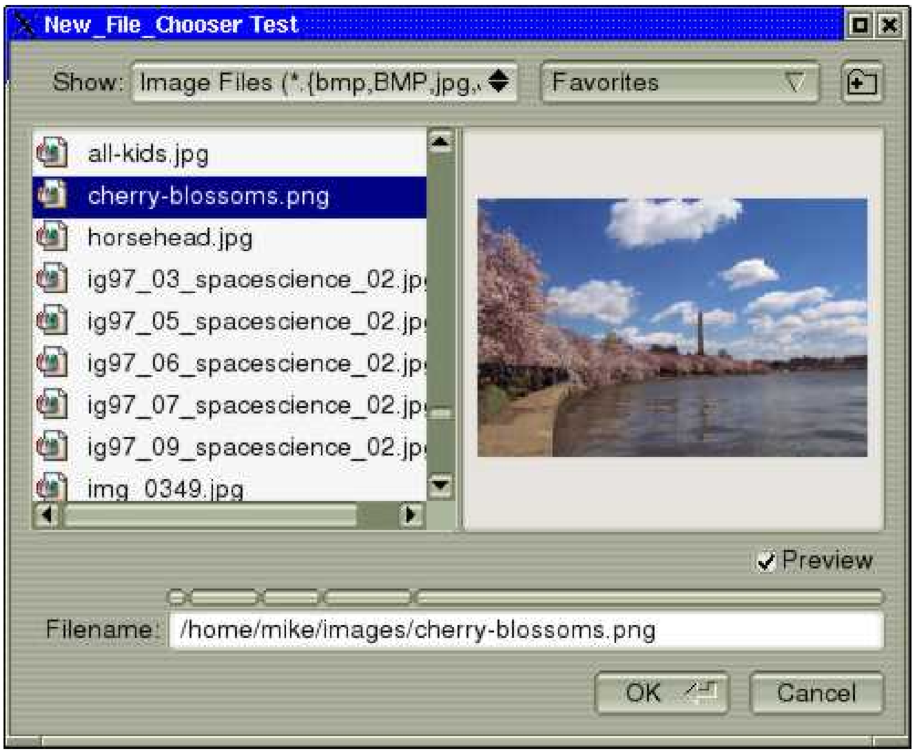
  Shows a file chooser dialog and gets a filename.
- void fl_file_chooser_callback (void(∗cb)(const char ∗))

  Set the file chooser callback.
- void fl_file_chooser_ok_label (const char ∗l)

  Set the ”OK” button label.

31.29.1 Detailed Description

The Fl_File_Chooser widget displays a standard file selection dialog that supports various selection modes.



all-kids.jpg
cherry-blossoms.png
horsehead.jpg
ig97_03_spacescience_02.jp
ig97_05_spacescience_02.jp
ig97_06_spacescience_02.jp
ig97_07_spacescience_02.jp
ig97_09_spacescience_02.jp
img_0349.jpg

Filename: /home/mike/images/cherry-blossoms.png

Figure 31.12: Fl_File_Chooser

The Fl_File_Chooser widget transmits UTF-8 encoded filenames to its user. It is recommended to open files that may have non-ASCII names with the fl_fopen() or fl_open() utility functions that handle these names in a cross-platform way (whereas the standard fopen()/open() functions fail on the MSWindows platform to open files with a non-ASCII name).

The Fl_File_Chooser class also exports several static values that may be used to localize or customize the appearance of all file chooser dialogs:

Member	Default value
add_favorites_label	"Add to Favorites"
all_files_label	"All Files (∗)"
custom_filter_label	"Custom Filter"
existing_file_label	"Please choose an existing file!"
favorites_label	"Favorites"
filename_label	"Filename:"
filesystems_label	"My Computer" (WIN32) "File Systems" (all others)
hidden_label	"Show hidden files:"
manage_favorites_label	"Manage Favorites"
new_directory_label	"New Directory?"
new_directory_tooltip	"Create a new directory."
preview_label	"Preview"
save_label	"Save"
show_label	"Show:"
sort	fl_numericsort

The Fl_File_Chooser::sort member specifies the sort function that is used when loading the contents of a directory and can be customized at run-time.

The Fl_File_Chooser class also exports the Fl_File_Chooser::newButton and Fl_File_Chooser::previewButton widgets so that application developers can control their appearance and use. For more complex customization, consider copying the FLTK file chooser code and changing it accordingly.

31.29.2 Constructor & Destructor Documentation

Fl_File_Chooser::Fl_File_Chooser (const char ∗ *pathname*, const char ∗ *pattern*, int *type*, const char ∗ *title*)

The constructor creates the Fl_File_Chooser dialog shown.

The pathname argument can be a directory name or a complete file name (in which case the corresponding file is highlighted in the list and in the filename input field.)

The pattern argument can be a NULL string or "∗" to list all files, or it can be a series of descriptions and filter strings separated by tab characters (\t). The format of filters is either "Description text (patterns)" or just "patterns". A file chooser that provides filters for HTML and image files might look like:

```
"HTML Files (*.html)\tImage Files (*.{bmp,gif,jpg,png})"
```

The file chooser will automatically add the "All Files (∗)" pattern to the end of the string you pass if you do not provide one. The first filter in the string is the default filter.

See the FLTK documentation on fl_filename_match() for the kinds of pattern strings that are supported.

The type argument can be one of the following:

- `SINGLE` - allows the user to select a single, existing file.
- `MULTI` - allows the user to select one or more existing files.
- `CREATE` - allows the user to select a single, existing file or specify a new filename.
- `DIRECTORY` - allows the user to select a single, existing directory.

The title argument is used to set the title bar text for the Fl_File_Chooser window.

Fl_File_Chooser::∼Fl_File_Chooser ( )

Destroys the widget and frees all memory used by it.

31.29.3 Member Function Documentation

Fl_Widget ∗ Fl_File_Chooser::add_extra (Fl_Widget ∗ *gr*)

Adds extra widget at the bottom of Fl_File_Chooser window.

Returns pointer for previous extra widget or NULL if not set previously. If argument is NULL only remove previous extra widget.

Note

> Fl_File_Chooser does **not** delete extra widget in destructor! To prevent memory leakage, don't forget to delete unused extra widgets

void Fl_File_Chooser::color (Fl_Color *c*)

Sets the background color of the Fl_File_Browser list.

Fl_Color Fl_File_Chooser::color ( )

Gets the background color of the Fl_File_Browser list.

int Fl_File_Chooser::count ( )

Returns the number of selected files.

void Fl_File_Chooser::directory (const char ∗ *pathname*)

Sets the current directory.

const char ∗ Fl_File_Chooser::directory ( )

Gets the current directory.

void Fl_File_Chooser::filter (const char ∗ *pattern*)

Sets or gets the current filename filter patterns.

The filter patterns use fl_filename_match(). Multiple patterns can be used by separating them with tabs, like `"∗.jpg\t∗.png\t∗.gif\t∗"`. In addition, you can provide human-readable labels with the patterns inside parenthesis, like `"JPEG Files (∗.jpg)\tPNG Files (∗.png)\tGIF Files (∗.gif)\tAll Files (∗)"` .

Use filter(NULL) to show all files.

int Fl_File_Chooser::filter_value ( )

Gets the current filename filter selection.

void Fl_File_Chooser::filter_value (int *f*)

Sets the current filename filter selection.

void Fl_File_Chooser::hide ( )

Hides the Fl_File_Chooser window.

void Fl_File_Chooser::iconsize (uchar *s*)

Sets the size of the icons in the Fl_File_Browser.
By default the icon size is set to 1.5 times the textsize().

uchar Fl_File_Chooser::iconsize ( )

Gets the size of the icons in the Fl_File_Browser.
By default the icon size is set to 1.5 times the textsize().

void Fl_File_Chooser::label (const char ∗ *l*)

Sets the title bar text for the Fl_File_Chooser.

const char ∗ Fl_File_Chooser::label ( )

Gets the title bar text for the Fl_File_Chooser.

void Fl_File_Chooser::preview (int *e*)

Enable or disable the preview tile.
1 = enable preview, 0 = disable preview.

int Fl_File_Chooser::preview ( ) const `[inline]`

Returns the current state of the preview box.

void Fl_File_Chooser::rescan ( )

Reloads the current directory in the Fl_File_Browser.

void Fl_File_Chooser::show ( )

Shows the Fl_File_Chooser window.

void Fl_File_Chooser::textcolor (Fl_Color *c*)

Sets the current Fl_File_Browser text color.

Fl_Color Fl_File_Chooser::textcolor ( )

Gets the current Fl_File_Browser text color.

void Fl_File_Chooser::textfont (Fl_Font *f*)

Sets the current Fl_File_Browser text font.

Fl_Font Fl_File_Chooser::textfont ( )

Gets the current Fl_File_Browser text font.

void Fl_File_Chooser::textsize (Fl_Fontsize *s*)

Sets the current Fl_File_Browser text size.

Fl_Fontsize Fl_File_Chooser::textsize ( )

Gets the current Fl_File_Browser text size.

void Fl_File_Chooser::type (int *t*)

Sets the current type of Fl_File_Chooser.

int Fl_File_Chooser::type ( )

Gets the current type of Fl_File_Chooser.

const char ∗ Fl_File_Chooser::value (int *f* = 1)

Gets the current value of the selected file(s).

`f` is a `1-based` index into a list of file names. The number of selected files is returned by Fl_File_-Chooser::count().

This sample code loops through all selected files:

```
// Get list of filenames user selected from a MULTI chooser
for ( int t=1; t<=chooser->count(); t++ ) {
const char *filename = chooser->value(t);
...
}
```

int Fl_File_Chooser::visible ( )

Returns 1 if the Fl_File_Chooser window is visible.

31.29.4 Member Data Documentation

Fl_File_Chooser::showHiddenButton

When checked, hidden files (i.e., filename begins with dot) are displayed.

The "showHiddenButton" button is exported so that application developers can control its appearance.

The documentation for this class was generated from the following files:

- Fl_File_Chooser.H
- Fl_File_Chooser.cxx
- Fl_File_Chooser2.cxx
- fl_file_dir.cxx

31.30 Fl_File_Icon Class Reference

The Fl_File_Icon class manages icon images that can be used as labels in other widgets and as icons in the FileBrowser widget.

`#include <Fl_File_Icon.H>`

Public Types

- enum {
  ANY, **PLAIN**, **FIFO**, **DEVICE**,
  LINK, **DIRECTORY** }
- enum {
  END, **COLOR**, **LINE**, **CLOSEDLINE**,
  POLYGON, **OUTLINEPOLYGON**, **VERTEX** }

Public Member Functions

- short ∗ add (short d)

  Adds a keyword value to the icon array, returning a pointer to it.
- short ∗ add_color (Fl_Color c)

  Adds a color value to the icon array, returning a pointer to it.
- short ∗ add_vertex (int x, int y)

  Adds a vertex value to the icon array, returning a pointer to it.
- short ∗ add_vertex (float x, float y)

  Adds a vertex value to the icon array, returning a pointer to it.
- void clear ()

  Clears all icon data from the icon.
- void draw (int x, int y, int w, int h, Fl_Color ic, int active=1)

  Draws an icon in the indicated area.
- Fl_File_Icon (const char ∗p, int t, int nd=0, short ∗d=0)

  Creates a new Fl_File_Icon with the specified information.
- void label (Fl_Widget ∗w)

  Applies the icon to the widget, registering the Fl_File_Icon label type as needed.
- void load (const char ∗f)

  Loads the specified icon image.
- int load_fti (const char ∗fti)

  Loads an SGI icon file.
- int load_image (const char ∗i)

  Load an image icon file from an image filename.
- Fl_File_Icon ∗ next ()

  Returns next file icon object.
- const char ∗ pattern ()

  Returns the filename matching pattern for the icon.
- int size ()

  Returns the number of words of data used by the icon.
- int type ()

  Returns the filetype associated with the icon, which can be one of the following:
- short ∗ value ()

  Returns the data array for the icon.
- ∼Fl_File_Icon ()

  The destructor destroys the icon and frees all memory that has been allocated for it.

Static Public Member Functions

- static Fl_File_Icon ∗ find (const char ∗filename, int filetype=ANY)

  Finds an icon that matches the given filename and file type.
- static Fl_File_Icon ∗ first ()

  Returns a pointer to the first icon in the list.
- static void labeltype (const Fl_Label ∗o, int x, int y, int w, int h, Fl_Align a)

  Draw the icon label.
- static void load_system_icons (void)

  Loads all system-defined icons.

31.30.1 Detailed Description

The Fl_File_Icon class manages icon images that can be used as labels in other widgets and as icons in the FileBrowser widget.

31.30.2 Constructor & Destructor Documentation

Fl_File_Icon::Fl_File_Icon (const char ∗ *p*, int *t*, int *nd* = 0, short ∗ *d* = 0)

Creates a new Fl_File_Icon with the specified information.
Parameters

`in`	*p*	filename pattern
`in`	*t*	file type
`in`	*nd*	number of data values
`in`	*d*	data values

31.30.3 Member Function Documentation

short ∗ Fl_File_Icon::add (short *d*)

Adds a keyword value to the icon array, returning a pointer to it.
Parameters

`in`	*d*	data value

short∗ Fl_File_Icon::add_color (Fl_Color *c*) `[inline]`

Adds a color value to the icon array, returning a pointer to it.
Parameters

`in`	*c*	color value

short∗ Fl_File_Icon::add_vertex (int *x*, int *y*) `[inline]`

Adds a vertex value to the icon array, returning a pointer to it.
The integer version accepts coordinates from 0 to 10000. The origin (0.0) is in the lower-lefthand corner of the icon.
Parameters

`in`	*x,y*	vertex coordinates

short∗ Fl_File_Icon::add_vertex (float *x*, float *y*) `[inline]`

Adds a vertex value to the icon array, returning a pointer to it.
The floating point version goes from 0.0 to 1.0. The origin (0.0) is in the lower-lefthand corner of the icon.
Parameters

`in`	*x,y*	vertex coordinates

void Fl_File_Icon::clear ( ) `[inline]`

Clears all icon data from the icon.

void Fl_File_Icon::draw (int *x*, int *y*, int *w*, int *h*, Fl_Color *ic*, int *active* = `1`)

Draws an icon in the indicated area.

Parameters

in	*x,y,w,h*	position and size
in	*ic*	icon color
in	*active*	status, default is active [non-zero]

Fl_File_Icon ∗ Fl_File_Icon::find (const char ∗*filename*, int *filetype* = *ANY*) `[static]`

Finds an icon that matches the given filename and file type.
Parameters

in	*filename*	name of file
in	*filetype*	enumerated file type

Returns

matching file icon or NULL

static Fl_File_Icon∗ Fl_File_Icon::first ( ) `[inline]`, `[static]`

Returns a pointer to the first icon in the list.

void Fl_File_Icon::label (Fl_Widget ∗ *w*)

Applies the icon to the widget, registering the Fl_File_Icon label type as needed.
Parameters

in	*w*	widget for which this icon will become the label

void Fl_File_Icon::labeltype (const Fl_Label ∗ *o*, int *x*, int *y*, int *w*, int *h*, Fl_Align *a*) `[static]`

Draw the icon label.
Parameters

in	*o*	label data
in	*x,y,w,h*	position and size of label
in	*a*	label alignment [not used]

void Fl_File_Icon::load (const char ∗ *f*)

Loads the specified icon image.
The format is deduced from the filename.
Parameters

in	*f*	filename

int Fl_File_Icon::load_fti (const char ∗ *fti*)

Loads an SGI icon file.

Parameters

in	*fti*	icon filename

Returns

0 on success, non-zero on error

int Fl_File_Icon::load_image (const char * *ifile*)

Load an image icon file from an image filename.
Parameters

in	*ifile*	image filename

Returns

0 on success, non-zero on error

void Fl_File_Icon::load_system_icons (void) [static]

Loads all system-defined icons.

This call is useful when using the FileChooser widget and should be used when the application starts:

```
Fl_File_Icon::load_system_icons();
```

Fl_File_Icon* Fl_File_Icon::next ( ) [inline]

Returns next file icon object.

See Fl_File_Icon::first()

const char* Fl_File_Icon::pattern ( ) [inline]

Returns the filename matching pattern for the icon.

int Fl_File_Icon::size ( ) [inline]

Returns the number of words of data used by the icon.

int Fl_File_Icon::type ( ) [inline]

Returns the filetype associated with the icon, which can be one of the following:

- Fl_File_Icon::ANY, any kind of file.
- Fl_File_Icon::PLAIN, plain files.
- Fl_File_Icon::FIFO, named pipes.
- Fl_File_Icon::DEVICE, character and block devices.
- Fl_File_Icon::LINK, symbolic links.
- Fl_File_Icon::DIRECTORY, directories.

short∗ Fl_File_Icon::value ( ) `[inline]`

Returns the data array for the icon.

The documentation for this class was generated from the following files:

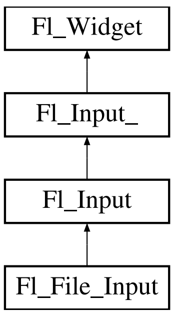
- Fl_File_Icon.H
- Fl_File_Icon.cxx
- Fl_File_Icon2.cxx

31.31 Fl_File_Input Class Reference

This widget displays a pathname in a text input field.

`#include <Fl_File_Input.H>`

Inheritance diagram for Fl_File_Input:



Public Member Functions

- Fl_Boxtype down_box () const

  Gets the box type used for the navigation bar.
- void down_box (Fl_Boxtype b)

  Sets the box type to use for the navigation bar.
- Fl_Color errorcolor () const

  Gets the current error color.
- void errorcolor (Fl_Color c)

  Sets the current error color to `c`*.*
- Fl_File_Input (int X, int Y, int W, int H, const char ∗L=0)

  Creates a new Fl_File_Input widget using the given position, size, and label string.
- virtual int handle (int event)

  Handle events in the widget.
- int value (const char ∗str)

  Sets the value of the widget given a new string value.
- int value (const char ∗str, int len)

  Sets the value of the widget given a new string value and its length.
- const char ∗ value ()

  Returns the current value, which is a pointer to an internal buffer and is valid only until the next event is handled.

Protected Member Functions

- virtual void draw ()

  Draws the file input widget.

Additional Inherited Members

31.31.1 Detailed Description

This widget displays a pathname in a text input field.

A navigation bar located above the input field allows the user to navigate upward in the directory tree. You may want to handle FL_WHEN_CHANGED events for tracking text changes and also FL_WHEN_RELEASE for button release when changing to parent dir. FL_WHEN_RELEASE callback won't be called if the directory clicked is the same as the current one.

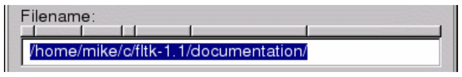


Figure 31.13: Fl_File_Input

Note

As all Fl_Input derived objects, Fl_File_Input may call its callback when losing focus (see FL_UNFOCUS) to update its state like its cursor shape. One resulting side effect is that you should call clear_changed() early in your callback to avoid reentrant calls if you plan to show another window or dialog box in the callback.

31.31.2 Constructor & Destructor Documentation

Fl_File_Input::Fl_File_Input (int *X*, int *Y*, int *W*, int *H*, const char ∗ *L* = `0`)

Creates a new Fl_File_Input widget using the given position, size, and label string.

The default boxtype is FL_DOWN_BOX.

Parameters

`in`	*X,Y,W,H*	position and size of the widget
`in`	*L*	widget label, default is no label

31.31.3 Member Function Documentation

Fl_Boxtype Fl_File_Input::down_box ( ) const `[inline]`

Gets the box type used for the navigation bar.

void Fl_File_Input::down_box (Fl_Boxtype *b*) `[inline]`

Sets the box type to use for the navigation bar.

Fl_Color Fl_File_Input::errorcolor ( ) const `[inline]`

Gets the current error color.

Todo Better docs for Fl_File_Input::errorcolor() - is it even used?

int Fl_File_Input::handle (int *event*) `[virtual]`

Handle events in the widget.

Return non zero if event is handled.

Parameters

`in`	*event*	

Reimplemented from Fl_Widget.

int Fl_File_Input::value (const char ∗ *str*)

Sets the value of the widget given a new string value.

Returns non 0 on success.

Parameters

`in`	*str*	new string value

int Fl_File_Input::value (const char ∗ *str,* int *len*)

Sets the value of the widget given a new string value and its length.

Returns non 0 on success.

Parameters

`in`	*str*	new string value
`in`	*len*	lengh of value

The documentation for this class was generated from the following files:

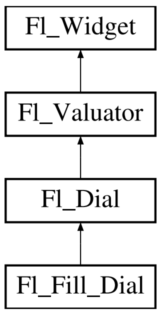
- Fl_File_Input.H
- Fl_File_Input.cxx

31.32 Fl_Fill_Dial Class Reference

Draws a dial with a filled arc.

`#include <Fl_Fill_Dial.H>`

Inheritance diagram for Fl_Fill_Dial:



Public Member Functions

- Fl_Fill_Dial (int X, int Y, int W, int H, const char ∗L)

  Creates a filled dial, also setting its type to FL_FILL_DIAL.

Additional Inherited Members

31.32.1 Detailed Description

Draws a dial with a filled arc.

31.32.2 Constructor & Destructor Documentation

Fl_Fill_Dial::Fl_Fill_Dial (int *X*, int *Y*, int *W*, int *H*, const char ∗ *L*)

Creates a filled dial, also setting its type to FL_FILL_DIAL.
The documentation for this class was generated from the following files:

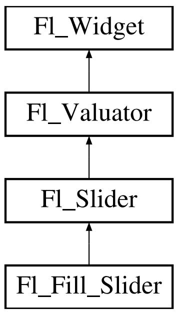
- Fl_Fill_Dial.H
- Fl_Dial.cxx

31.33 Fl_Fill_Slider Class Reference

Widget that draws a filled horizontal slider, useful as a progress or value meter.

```
#include <Fl_Fill_Slider.H>
```

Inheritance diagram for Fl_Fill_Slider:



Public Member Functions

- Fl_Fill_Slider (int X, int Y, int W, int H, const char ∗L=0)

  Creates the slider from its position,size and optional title.

Additional Inherited Members

31.33.1 Detailed Description

Widget that draws a filled horizontal slider, useful as a progress or value meter.

31.33.2 Constructor & Destructor Documentation

Fl_Fill_Slider::Fl_Fill_Slider (int *X*, int *Y*, int *W*, int *H*, const char ∗ *L* = `0`)

Creates the slider from its position,size and optional title.
The documentation for this class was generated from the following files:

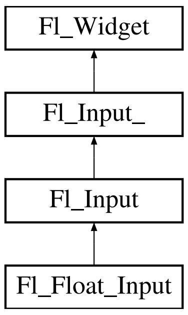
- Fl_Fill_Slider.H
- Fl_Slider.cxx

31.34 Fl_Float_Input Class Reference

The Fl_Float_Input class is a subclass of Fl_Input that only allows the user to type floating point numbers (sign, digits, decimal point, more digits, 'E' or 'e', sign, digits).

```
#include <Fl_Float_Input.H>
```

Inheritance diagram for Fl_Float_Input:



Public Member Functions

- Fl_Float_Input (int X, int Y, int W, int H, const char ∗l=0)

  Creates a new Fl_Float_Input widget using the given position, size, and label string.

Additional Inherited Members

31.34.1 Detailed Description

The Fl_Float_Input class is a subclass of Fl_Input that only allows the user to type floating point numbers (sign, digits, decimal point, more digits, 'E' or 'e', sign, digits).

31.34.2 Constructor & Destructor Documentation

Fl_Float_Input::Fl_Float_Input (int *X*, int *Y*, int *W*, int *H*, const char ∗ *l* = *0*)

Creates a new Fl_Float_Input widget using the given position, size, and label string.

The default boxtype is FL_DOWN_BOX.

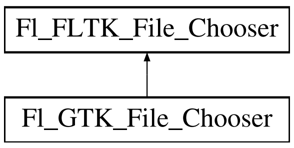
Inherited destructor destroys the widget and any value associated with it.

The documentation for this class was generated from the following files:

- Fl_Float_Input.H
- Fl_Input.cxx

31.35 Fl_FLTK_File_Chooser Class Reference

Inheritance diagram for Fl_FLTK_File_Chooser:



Protected Member Functions

- virtual int **count** () const
- void **directory** (const char ∗val)
- const char ∗ **directory** () const
- void **errmsg** (const char ∗msg)
- const char ∗ **errmsg** () const
- int **exist_dialog** ()
- virtual const char ∗ **filename** () const
- virtual const char ∗ **filename** (int i) const
- const char ∗ **filter** () const
- void **filter** (const char ∗)
- void **filter_value** (int i)
- int **filter_value** () const
- int **filters** () const
- **Fl_FLTK_File_Chooser** (int val)
- void **options** (int)
- int **options** () const
- void **parse_filter** ()
- void **preset_file** (const char ∗)
- const char ∗ **preset_file** () const
- virtual int **show** ()
- virtual void **title** (const char ∗)
- virtual const char ∗ **title** () const
- virtual void **type** (int)
- int **type** () const
- int **type_fl_file** (int)

Protected Attributes

- int **_btype**
- char ∗ **_directory**
- char ∗ **_errmsg**
- Fl_File_Chooser ∗ **_file_chooser**
- char ∗ **_filter**
- int **_filtvalue**
- int **_nfilters**
- int **_options**
- char ∗ **_parsedfilt**
- char ∗ **_preset_file**
- char ∗ **_prevvalue**

Friends

- class **Fl_Native_File_Chooser**

The documentation for this class was generated from the following files:

- Fl_Native_File_Chooser.H
- Fl_Native_File_Chooser_FLTK.cxx

31.36 Fl_Font_Descriptor Class Reference

This a structure for an actual system font, with junk to help choose it and info on character sizes.

`#include <Fl_Font.H>`

Public Attributes

- Fl_Font_Descriptor ∗ next

  linked list for this Fl_Fontdesc
- Fl_Fontsize size

  font size

31.36.1 Detailed Description

This a structure for an actual system font, with junk to help choose it and info on character sizes.

Each Fl_Fontdesc has a linked list of these. These are created the first time each system font/size combination is used.

The documentation for this class was generated from the following file:

- Fl_Font.H

31.37 Fl_Fontdesc Struct Reference

Public Attributes

- Fl_Font_Descriptor ∗ **first**
- char **fontname** [128]
- int **n**
- const char ∗ **name**
- char ∗∗ **xlist**

The documentation for this struct was generated from the following file:

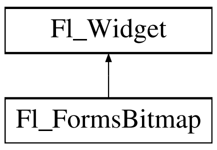
- Fl_Font.H

31.38 Fl_FormsBitmap Class Reference

Forms compatibility Bitmap Image Widget.

`#include <Fl_FormsBitmap.H>`

Inheritance diagram for Fl_FormsBitmap:



Public Member Functions

- void bitmap (Fl_Bitmap ∗B)

  Sets a new bitmap.
- Fl_Bitmap ∗ bitmap () const

  Gets a the current associated Fl_Bitmap objects.
- Fl_FormsBitmap (Fl_Boxtype, int, int, int, int, const char ∗=0)

  Creates a bitmap widget from a box type, position, size and optional label specification.
- void set (int W, int H, const uchar ∗bits)

  Sets a new bitmap bits with size W,H.

Protected Member Functions

- void draw ()

  Draws the bitmap and its associated box.

Additional Inherited Members

31.38.1 Detailed Description

Forms compatibility Bitmap Image Widget.

31.38.2 Member Function Documentation

void Fl_FormsBitmap::bitmap (Fl_Bitmap ∗ *B*) `[inline]`

Sets a new bitmap.

Fl_Bitmap∗ Fl_FormsBitmap::bitmap ( ) const `[inline]`

Gets a the current associated Fl_Bitmap objects.

void Fl_FormsBitmap::draw (void) `[protected], [virtual]`

Draws the bitmap and its associated box.
Implements Fl_Widget.

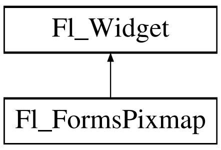
void Fl_FormsBitmap::set (int *W*, int *H*, const uchar ∗ *bits*)

Sets a new bitmap bits with size W,H.
Deletes the previous one.
The documentation for this class was generated from the following files:

- Fl_FormsBitmap.H
- forms_bitmap.cxx

31.39 Fl_FormsPixmap Class Reference

Forms pixmap drawing routines.
`#include <Fl_FormsPixmap.H>`
Inheritance diagram for Fl_FormsPixmap:



Public Member Functions

- Fl_FormsPixmap (Fl_Boxtype t, int X, int Y, int W, int H, const char ∗L=0)

  Creates a new Fl_FormsPixmap widget using the given box type, position, size and label string.
- void Pixmap (Fl_Pixmap ∗B)

  Set the internal pixmap pointer to an existing pixmap.
- Fl_Pixmap ∗ Pixmap () const

  Get the internal pixmap pointer.
- void set (char ∗const ∗bits)

  Set/create the internal pixmap using raw data.

Protected Member Functions

- void draw ()

  Draws the widget.

Additional Inherited Members

31.39.1 Detailed Description

Forms pixmap drawing routines.

31.39.2 Constructor & Destructor Documentation

Fl_FormsPixmap::Fl_FormsPixmap (Fl_Boxtype *t*, int *X*, int *Y*, int *W*, int *H*, const char ∗ *L* = 0)

Creates a new Fl_FormsPixmap widget using the given box type, position, size and label string.
Parameters

in	*t*	box type
in	*X,Y,W,H*	position and size
in	*L*	widget label, default is no label

31.39.3 Member Function Documentation

void Fl_FormsPixmap::draw ( ) [protected], [virtual]

Draws the widget.
Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call redraw() instead.
Override this function to draw your own widgets.
If you ever need to call another widget's draw method *from within your own draw() method*, e.g. for an embedded scrollbar, you can do it (because draw() is virtual) like this:

```
Fl_Widget *s = &scroll;                  // scroll is an embedded Fl_Scrollbar
s->draw();                          // calls Fl_Scrollbar::draw()
```

Implements Fl_Widget.

void Fl_FormsPixmap::Pixmap (Fl_Pixmap ∗ *B*) `[inline]`

Set the internal pixmap pointer to an existing pixmap.

Parameters

in	*B*	existing pixmap

Fl_Pixmap∗ Fl_FormsPixmap::Pixmap ( ) const `[inline]`

Get the internal pixmap pointer.

void Fl_FormsPixmap::set (char ∗const ∗ *bits*)

Set/create the internal pixmap using raw data.
Parameters

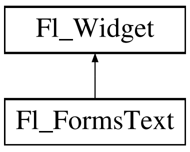
in	*bits*	raw data

The documentation for this class was generated from the following files:

- Fl_FormsPixmap.H
- forms_pixmap.cxx

31.40 Fl_FormsText Class Reference

Inheritance diagram for Fl_FormsText:



Public Member Functions

- **Fl_FormsText** (Fl_Boxtype b, int X, int Y, int W, int H, const char ∗l=0)

Protected Member Functions

- void draw ()

  Draws the widget.

Additional Inherited Members

31.40.1 Member Function Documentation

void Fl_FormsText::draw ( ) `[protected], [virtual]`

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call redraw() instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw() method*, e.g. for an embedded scrollbar, you can do it (because draw() is virtual) like this:

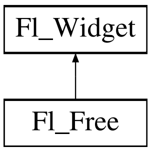
```
Fl_Widget *s = &scroll;                // scroll is an embedded Fl_Scrollbar
s->draw();                  // calls Fl_Scrollbar::draw()
```

Implements Fl_Widget.
The documentation for this class was generated from the following file:

- forms.H

31.41 Fl_Free Class Reference

Emulation of the Forms "free" widget.
`#include <Fl_Free.H>`
Inheritance diagram for Fl_Free:



Public Member Functions

- Fl_Free (uchar t, int X, int Y, int W, int H, const char *L, FL_HANDLEPTR hdl)

  Create a new Fl_Free widget with type, position, size, label and handler.
- int handle (int e)

  Handles the specified event.
- ~Fl_Free ()

  The destructor will call the handle function with the event FL_FREE_MEM.

Protected Member Functions

- void draw ()

  Draws the widget.

Additional Inherited Members

31.41.1 Detailed Description

Emulation of the Forms "free" widget.
This emulation allows the free demo to run, and appears to be useful for porting programs written in Forms which use the free widget or make subclasses of the Forms widgets.
There are five types of free, which determine when the handle function is called:

- `FL_NORMAL_FREE` normal event handling.
- `FL_SLEEPING_FREE` deactivates event handling (widget is inactive).
- `FL_INPUT_FREE` accepts FL_FOCUS events.
- `FL_CONTINUOUS_FREE` sets a timeout callback 100 times a second and provides an FL_STEP event. This has obvious detrimental effects on machine performance.
- `FL_ALL_FREE` same as FL_INPUT_FREE and FL_CONTINUOUS_FREE.

31.41.2 Constructor & Destructor Documentation

Fl_Free::Fl_Free (uchar *t*, int *X*, int *Y*, int *W*, int *H*, const char * *L*, FL_HANDLEPTR *hdl*)

Create a new Fl_Free widget with type, position, size, label and handler.

Parameters

in	*t*	type
in	*X,Y,W,H*	position and size
in	*L*	widget label
in	*hdl*	handler function

The constructor takes both the type and the handle function. The handle function should be declared as follows:

```
int handle_function(Fl_Widget *w,
                    int       event,
                      float     event_x,
                      float     event_y,
                      char      key)
```

This function is called from the handle() method in response to most events, and is called by the draw() method.

The event argument contains the event type:

```
// old event names for compatibility:
#define FL_MOUSE                FL_DRAG
#define FL_DRAW         0
#define FL_STEP         9
#define FL_FREEMEM              12
#define FL_FREEZE               FL_UNMAP
#define FL_THAW         FL_MAP
```

31.41.3 Member Function Documentation

void Fl_Free::draw ( ) [protected], [virtual]

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call redraw() instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw() method*, e.g. for an embedded scrollbar, you can do it (because draw() is virtual) like this:

```
Fl_Widget *s = &scroll;               // scroll is an embedded Fl_Scrollbar
s->draw();                    // calls Fl_Scrollbar::draw()
```

Implements Fl_Widget.

int Fl_Free::handle (int *event*) [virtual]

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited handle() method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters

in	*event*	the kind of event received

Return values

0	if the event was not used or understood
1	if the event was used and can be deleted

See Also

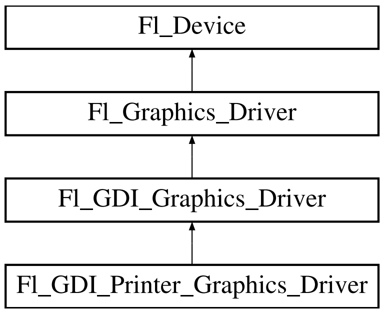
Fl_Event

Reimplemented from Fl_Widget.
The documentation for this class was generated from the following files:

- Fl_Free.H
- forms_free.cxx

31.42 Fl_GDI_Graphics_Driver Class Reference

The MSWindows-specific graphics class.
`#include <Fl_Device.H>`
Inheritance diagram for Fl_GDI_Graphics_Driver:



Public Member Functions

- const char ∗ class_name ()

  Returns the name of the class of this object.
- void color (Fl_Color c)

  see fl_color(Fl_Color c).
- void color (uchar r, uchar g, uchar b)

  see fl_color(uchar r, uchar g, uchar b).
- void copy_offscreen (int x, int y, int w, int h, Fl_Offscreen pixmap, int srcx, int srcy)

  see fl_copy_offscreen()
- int descent ()

  see fl_descent().
- void draw (const char ∗str, int n, int x, int y)

  see fl_draw(const char ∗str, int n, int x, int y).
- void draw (int angle, const char ∗str, int n, int x, int y)

  see fl_draw(int angle, const char ∗str, int n, int x, int y).
- void draw (Fl_Pixmap ∗pxm, int XP, int YP, int WP, int HP, int cx, int cy)

  Draws an Fl_Pixmap object to the device.
- void draw (Fl_Bitmap ∗pxm, int XP, int YP, int WP, int HP, int cx, int cy)

Draws an Fl_Bitmap object to the device.

- void draw (Fl_RGB_Image ∗img, int XP, int YP, int WP, int HP, int cx, int cy)

  Draws an Fl_RGB_Image object to the device.
- void draw_image (const uchar ∗buf, int X, int Y, int W, int H, int D=3, int L=0)

  see fl_draw_image(const uchar∗ buf, int X,int Y,int W,int H, int D, int L).
- void draw_image (Fl_Draw_Image_Cb cb, void ∗data, int X, int Y, int W, int H, int D=3)

  see fl_draw_image(Fl_Draw_Image_Cb cb, void∗ data, int X,int Y,int W,int H, int D).
- void draw_image_mono (const uchar ∗buf, int X, int Y, int W, int H, int D=1, int L=0)

  see fl_draw_image_mono(const uchar∗ buf, int X,int Y,int W,int H, int D, int L).
- void draw_image_mono (Fl_Draw_Image_Cb cb, void ∗data, int X, int Y, int W, int H, int D=1)

  see fl_draw_image_mono(Fl_Draw_Image_Cb cb, void∗ data, int X,int Y,int W,int H, int D).
- void font (Fl_Font face, Fl_Fontsize size)

  see fl_font(Fl_Font face, Fl_Fontsize size).
- int height ()

  see fl_height().
- void rtl_draw (const char ∗str, int n, int x, int y)

  see fl_rtl_draw(const char ∗str, int n, int x, int y).
- void text_extents (const char ∗, int n, int &dx, int &dy, int &w, int &h)

  see fl_text_extents(const char∗, int n, int& dx, int& dy, int& w, int& h).
- double width (const char ∗str, int n)

  see fl_width(const char ∗str, int n).
- double width (unsigned int c)

  see fl_width(unsigned int n).

Static Public Attributes

- static const char ∗ **class_id** = "Fl_GDI_Graphics_Driver"

Additional Inherited Members

31.42.1 Detailed Description

The MSWindows-specific graphics class.

This class is implemented only on the MSWindows platform.

31.42.2 Member Function Documentation

const char∗ Fl_GDI_Graphics_Driver::class_name ( ) [inline], [virtual]

Returns the name of the class of this object.

Use of the class_name() function is discouraged because it will be removed from future FLTK versions.

The class of an instance of an Fl_Device subclass can be checked with code such as:

```
if ( instance->class_name() == Fl_Printer::class_id ) { ... }
```

Reimplemented from Fl_Graphics_Driver.

Reimplemented in Fl_GDI_Printer_Graphics_Driver.

void Fl_GDI_Graphics_Driver::color (Fl_Color *c*) [virtual]

see fl_color(Fl_Color c).

Reimplemented from Fl_Graphics_Driver.

void Fl_GDI_Graphics_Driver::color (uchar *r*, uchar *g*, uchar *b*) `[virtual]`

see fl_color(uchar r, uchar g, uchar b).

Reimplemented from Fl_Graphics_Driver.

int Fl_GDI_Graphics_Driver::descent ( ) `[virtual]`

see fl_descent().

Reimplemented from Fl_Graphics_Driver.

void Fl_GDI_Graphics_Driver::draw (const char * *str*, int *n*, int *x*, int *y*) `[virtual]`

see fl_draw(const char *str, int n, int x, int y).

Reimplemented from Fl_Graphics_Driver.

void Fl_GDI_Graphics_Driver::draw (int *angle*, const char * *str*, int *n*, int *x*, int *y*) `[virtual]`

see fl_draw(int angle, const char *str, int n, int x, int y).

Reimplemented from Fl_Graphics_Driver.

void Fl_GDI_Graphics_Driver::draw (Fl_Pixmap * *pxm*, int *XP*, int *YP*, int *WP*, int *HP*, int *cx*, int *cy*) `[virtual]`

Draws an Fl_Pixmap object to the device.

Specifies a bounding box for the image, with the origin (upper left-hand corner) of the image offset by the cx and cy arguments.

Reimplemented from Fl_Graphics_Driver.

Reimplemented in Fl_GDI_Printer_Graphics_Driver.

void Fl_GDI_Graphics_Driver::draw (Fl_Bitmap * *bm*, int *XP*, int *YP*, int *WP*, int *HP*, int *cx*, int *cy*) `[virtual]`

Draws an Fl_Bitmap object to the device.

Specifies a bounding box for the image, with the origin (upper left-hand corner) of the image offset by the cx and cy arguments.

Reimplemented from Fl_Graphics_Driver.

Reimplemented in Fl_GDI_Printer_Graphics_Driver.

void Fl_GDI_Graphics_Driver::draw (Fl_RGB_Image * *rgb*, int *XP*, int *YP*, int *WP*, int *HP*, int *cx*, int *cy*) `[virtual]`

Draws an Fl_RGB_Image object to the device.

Specifies a bounding box for the image, with the origin (upper left-hand corner) of the image offset by the cx and cy arguments.

Reimplemented from Fl_Graphics_Driver.

void Fl_GDI_Graphics_Driver::draw_image (const uchar * *buf*, int *X*, int *Y*, int *W*, int *H*, int *D* = *3*, int *L* = *0*) `[virtual]`

see fl_draw_image(const uchar* buf, int X,int Y,int W,int H, int D, int L).

Reimplemented from Fl_Graphics_Driver.

void Fl_GDI_Graphics_Driver::draw_image (Fl_Draw_Image_Cb *cb*, void ∗ *data*, int *X*, int *Y*, int *W*, int *H*, int *D* = *3*) `[virtual]`

see fl_draw_image(Fl_Draw_Image_Cb cb, void∗ data, int X,int Y,int W,int H, int D).
Reimplemented from Fl_Graphics_Driver.

void Fl_GDI_Graphics_Driver::draw_image_mono (const uchar ∗ *buf*, int *X*, int *Y*, int *W*, int *H*, int *D* = *1*, int *L* = *0*) `[virtual]`

see fl_draw_image_mono(const uchar∗ buf, int X,int Y,int W,int H, int D, int L).
Reimplemented from Fl_Graphics_Driver.

void Fl_GDI_Graphics_Driver::draw_image_mono (Fl_Draw_Image_Cb *cb*, void ∗ *data*, int *X*, int *Y*, int *W*, int *H*, int *D* = *1*) `[virtual]`

see fl_draw_image_mono(Fl_Draw_Image_Cb cb, void∗ data, int X,int Y,int W,int H, int D).
Reimplemented from Fl_Graphics_Driver.

void Fl_GDI_Graphics_Driver::font (Fl_Font *face*, Fl_Fontsize *fsize*) `[virtual]`

see fl_font(Fl_Font face, Fl_Fontsize size).
Reimplemented from Fl_Graphics_Driver.

int Fl_GDI_Graphics_Driver::height ( ) `[virtual]`

see fl_height().
Reimplemented from Fl_Graphics_Driver.

void Fl_GDI_Graphics_Driver::rtl_draw (const char ∗ *str*, int *n*, int *x*, int *y*) `[virtual]`

see fl_rtl_draw(const char ∗str, int n, int x, int y).
Reimplemented from Fl_Graphics_Driver.

void Fl_GDI_Graphics_Driver::text_extents (const char ∗ *t*, int *n*, int & *dx*, int & *dy*, int & *w*, int & *h*) `[virtual]`

see fl_text_extents(const char∗, int n, int& dx, int& dy, int& w, int& h).
Reimplemented from Fl_Graphics_Driver.

double Fl_GDI_Graphics_Driver::width (const char ∗ *str*, int *n*) `[virtual]`

see fl_width(const char ∗str, int n).
Reimplemented from Fl_Graphics_Driver.

double Fl_GDI_Graphics_Driver::width (unsigned int *c*) `[virtual]`

see fl_width(unsigned int n).
Reimplemented from Fl_Graphics_Driver.
The documentation for this class was generated from the following files:

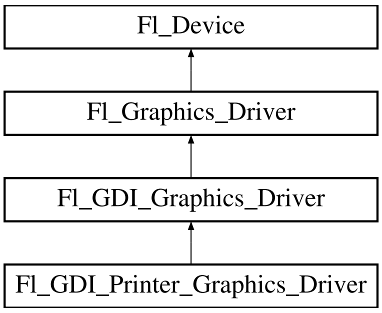
- Fl_Device.H
- fl_color_win32.cxx
- Fl_Device.cxx
- fl_draw_image_win32.cxx

31.43 Fl_GDI_Printer_Graphics_Driver Class Reference

The graphics driver used when printing on MSWindows.

`#include <Fl_Device.H>`

Inheritance diagram for Fl_GDI_Printer_Graphics_Driver:



Public Member Functions

- const char ∗ class_name ()

  Returns the name of the class of this object.
- void draw (Fl_Pixmap ∗pxm, int XP, int YP, int WP, int HP, int cx, int cy)

  Draws an Fl_Pixmap object to the device.
- void draw (Fl_Bitmap ∗bm, int XP, int YP, int WP, int HP, int cx, int cy)

  Draws an Fl_Bitmap object to the device.
- int draw_scaled (Fl_Image ∗img, int XP, int YP, int WP, int HP)

  Draws an Fl_Image scaled to width `W` & height `H` with top-left corner at X,Y.

Static Public Attributes

- static const char ∗ **class_id** = "Fl_GDI_Printer_Graphics_Driver"

Additional Inherited Members

31.43.1 Detailed Description

The graphics driver used when printing on MSWindows.

This class is implemented only on the MSWindows platform. It 's extremely similar to Fl_GDI_Graphics_Driver.

31.43.2 Member Function Documentation

const char∗ Fl_GDI_Printer_Graphics_Driver::class_name ( ) `[inline]`, `[virtual]`

Returns the name of the class of this object.

Use of the class_name() function is discouraged because it will be removed from future FLTK versions. The class of an instance of an Fl_Device subclass can be checked with code such as:

```
if ( instance->class_name() == Fl_Printer::class_id ) { ... }
```

Reimplemented from Fl_GDI_Graphics_Driver.

void Fl_GDI_Printer_Graphics_Driver::draw (Fl_Pixmap ∗ *pxm*, int *XP*, int *YP*, int *WP*, int *HP*, int *cx*, int *cy*) `[virtual]`

Draws an Fl_Pixmap object to the device.

Specifies a bounding box for the image, with the origin (upper left-hand corner) of the image offset by the cx and cy arguments.

Reimplemented from Fl_GDI_Graphics_Driver.

void Fl_GDI_Printer_Graphics_Driver::draw (Fl_Bitmap ∗ *bm*, int *XP*, int *YP*, int *WP*, int *HP*, int *cx*, int *cy*) `[virtual]`

Draws an Fl_Bitmap object to the device.

Specifies a bounding box for the image, with the origin (upper left-hand corner) of the image offset by the cx and cy arguments.

Reimplemented from Fl_GDI_Graphics_Driver.

int Fl_GDI_Printer_Graphics_Driver::draw_scaled (Fl_Image ∗ *img*, int *X*, int *Y*, int *W*, int *H*) `[virtual]`

Draws an Fl_Image scaled to width `W` & height `H` with top-left corner at *X*,Y.

Returns

zero when the graphics driver doesn't implement scaled drawing, non-zero if it does implement it.

Reimplemented from Fl_Graphics_Driver.

The documentation for this class was generated from the following files:

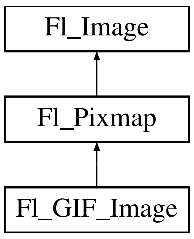
- Fl_Device.H
- Fl_Device.cxx

31.44 Fl_GIF_Image Class Reference

The Fl_GIF_Image class supports loading, caching, and drawing of Compuserve GIF^{SM} images.

`#include <Fl_GIF_Image.H>`

Inheritance diagram for Fl_GIF_Image:



Public Member Functions

- Fl_GIF_Image (const char ∗filename)

  The constructor loads the named GIF image.

Additional Inherited Members

31.44.1 Detailed Description

The Fl_GIF_Image class supports loading, caching, and drawing of Compuserve GIF^{SM} images.

The class loads the first image and supports transparency.

31.44.2 Constructor & Destructor Documentation

Fl_GIF_Image::Fl_GIF_Image (const char ∗ *infname*)

The constructor loads the named GIF image.

The destructor frees all memory and server resources that are used by the image.

Use Fl_Image::fail() to check if Fl_GIF_Image failed to load. fail() returns ERR_FILE_ACCESS if the file could not be opened or read, ERR_FORMAT if the GIF format could not be decoded, and ERR_NO_IMAGE if the image could not be loaded for another reason.

The documentation for this class was generated from the following files:

- Fl_GIF_Image.H
- Fl_GIF_Image.cxx

31.45 Fl_Gl_Choice Class Reference

Static Public Member Functions

- static Fl_Gl_Choice ∗ **find** (int mode, const int ∗)

Public Attributes

- Colormap **colormap**
- XVisualInfo ∗ **vis**

The documentation for this class was generated from the following files:

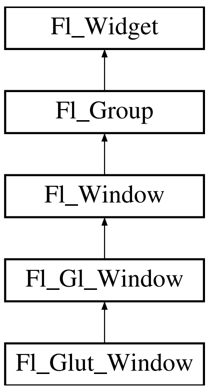
- Fl_Gl_Choice.H
- Fl_Gl_Choice.cxx

31.46 Fl_Gl_Window Class Reference

The Fl_Gl_Window widget sets things up so OpenGL works.

`#include <Fl_Gl_Window.H>`

Inheritance diagram for Fl_Gl_Window:



Public Member Functions

- virtual Fl_Gl_Window ∗ as_gl_window ()

  Returns an Fl_Gl_Window pointer if this widget is an Fl_Gl_Window.
- int can_do ()

  Returns non-zero if the hardware supports the current OpenGL mode.
- int can_do_overlay ()

  Returns true if the hardware overlay is possible.
- void ∗ context () const

  Returns a pointer to the GLContext that this window is using.
- void context (void ∗, int destroy_flag=0)

  Sets a pointer to the GLContext that this window is using.
- char context_valid () const

  Will only be set if the OpenGL context is created or recreated.
- void context_valid (char v)

  See char Fl_Gl_Window::context_valid() const.
- Fl_Gl_Window (int W, int H, const char ∗l=0)

  Creates a new Fl_Gl_Window widget using the given size, and label string.
- Fl_Gl_Window (int X, int Y, int W, int H, const char ∗l=0)

  Creates a new Fl_Gl_Window widget using the given position, size, and label string.
- void flush ()

  Forces the window to be drawn, this window is also made current and calls draw().
- int handle (int)

  Handle some FLTK events as needed.
- void hide ()

  Hides the window and destroys the OpenGL context.
- void hide_overlay ()

  Hides the window if it is not this window, does nothing in WIN32.
- void invalidate ()

  The invalidate() method turns off valid() and is equivalent to calling value(0).
- void make_current ()

  The make_current() method selects the OpenGL context for the widget.
- void make_overlay_current ()

  The make_overlay_current() method selects the OpenGL context for the widget's overlay.
- Fl_Mode mode () const

  Returns the current OpenGL capabilites of the window.
- int mode (int a)

  Set or change the OpenGL capabilites of the window.
- int mode (const int ∗a)

  Set the OpenGL capabilites of the window using platform-specific data.
- void ortho ()

  Sets the projection so 0,0 is in the lower left of the window and each pixel is 1 unit wide/tall.
- int pixel_h ()

  Gives the window height in OpenGL pixels.
- int pixel_w ()

  Gives the window width in OpenGL pixels.
- float pixels_per_unit ()

The number of pixels per FLTK unit of length for the window.

- void redraw_overlay ()

  This method causes draw_overlay() to be called at a later time.
- void resize (int, int, int, int)

  Changes the size and position of the window.
- void show ()

  Puts the window on the screen.
- void **show** (int a, char **b)
- void swap_buffers ()

  The swap_buffers() method swaps the back and front buffers.
- char valid () const

  Is turned off when FLTK creates a new context for this window or when the window resizes, and is turned on after draw() is called.
- void valid (char v)

  See char Fl_Gl_Window::valid() const.
- ∼Fl_Gl_Window ()

  The destructor removes the widget and destroys the OpenGL context associated with it.

Static Public Member Functions

- static int can_do (int m)

  Returns non-zero if the hardware supports the given OpenGL mode.
- static int can_do (const int *m)

  Returns non-zero if the hardware supports the given OpenGL mode.

Protected Member Functions

- virtual void draw ()

  Draws the Fl_Gl_Window.

Friends

- class **_Fl_Gl_Overlay**

Additional Inherited Members

31.46.1 Detailed Description

The Fl_Gl_Window widget sets things up so OpenGL works.

It also keeps an OpenGL "context" for that window, so that changes to the lighting and projection may be reused between redraws. Fl_Gl_Window also flushes the OpenGL streams and swaps buffers after draw() returns.

OpenGL hardware typically provides some overlay bit planes, which are very useful for drawing UI controls atop your 3D graphics. If the overlay hardware is not provided, FLTK tries to simulate the overlay. This works pretty well if your graphics are double buffered, but not very well for single-buffered.

Please note that the FLTK drawing and clipping functions will not work inside an Fl_Gl_Window. All drawing should be done using OpenGL calls exclusively. Even though Fl_Gl_Window is derived from Fl_Group, it is not useful to add other FLTK Widgets as children, unless those widgets are modified to draw using OpenGL calls.

31.46.2 Constructor & Destructor Documentation

Fl_Gl_Window::Fl_Gl_Window (int *W*, int *H*, const char ∗ *l* = 0) `[inline]`

Creates a new Fl_Gl_Window widget using the given size, and label string.

The default boxtype is FL_NO_BOX. The default mode is FL_RGB|FL_DOUBLE|FL_DEPTH.

Fl_Gl_Window::Fl_Gl_Window (int *X*, int *Y*, int *W*, int *H*, const char ∗ *l* = 0) `[inline]`

Creates a new Fl_Gl_Window widget using the given position, size, and label string.

The default boxtype is FL_NO_BOX. The default mode is FL_RGB|FL_DOUBLE|FL_DEPTH.

31.46.3 Member Function Documentation

virtual Fl_Gl_Window∗ Fl_Gl_Window::as_gl_window ( ) `[inline]`, `[virtual]`

Returns an Fl_Gl_Window pointer if this widget is an Fl_Gl_Window.

Use this method if you have a widget (pointer) and need to know whether this widget is derived from Fl_Gl_Window. If it returns non-NULL, then the widget in question is derived from Fl_Gl_Window.

Return values

NULL	if this widget is not derived from Fl_Gl_Window.

Note

This method is provided to avoid dynamic_cast.

See Also

Fl_Widget::as_group(), Fl_Widget::as_window()

Reimplemented from Fl_Widget.

static int Fl_Gl_Window::can_do (int *m*) `[inline]`, `[static]`

Returns non-zero if the hardware supports the given OpenGL mode.

static int Fl_Gl_Window::can_do (const int ∗ *m*) `[inline]`, `[static]`

Returns non-zero if the hardware supports the given OpenGL mode.

See Also

Fl_Gl_Window::mode(const int ∗a)

int Fl_Gl_Window::can_do ( ) `[inline]`

Returns non-zero if the hardware supports the current OpenGL mode.

int Fl_Gl_Window::can_do_overlay ( )

Returns true if the hardware overlay is possible.

If this is false, FLTK will try to simulate the overlay, with significant loss of update speed. Calling this will cause FLTK to open the display.

void∗ Fl_Gl_Window::context ( ) const `[inline]`

Returns a pointer to the GLContext that this window is using.

See Also

void context(void∗ v, int destroy_flag)

void Fl_Gl_Window::context (void ∗ *v*, int *destroy_flag* = `0`)

Sets a pointer to the GLContext that this window is using.

This is a system-dependent structure, but it is portable to copy the context from one window to another. You can also set it to NULL, which will force FLTK to recreate the context the next time make_current() is called, this is useful for getting around bugs in OpenGL implementations.

If *destroy_flag* is true the context will be destroyed by fltk when the window is destroyed, or when the mode() is changed, or the next time context(x) is called.

char Fl_Gl_Window::context_valid ( ) const `[inline]`

Will only be set if the OpenGL context is created or recreated.

It differs from Fl_Gl_Window::valid() which is also set whenever the context changes size.

void Fl_Gl_Window::draw (void) `[protected]`, `[virtual]`

Draws the Fl_Gl_Window.

You ***must*** *subclass* Fl_Gl_Window and provide an implementation for draw().

You ***must*** *override* the draw() method.

You may also provide an implementation of draw_overlay() if you want to draw into the overlay planes. You can avoid reinitializing the viewport and lights and other things by checking valid() at the start of draw() and only doing the initialization if it is false.

The draw() method can *only* use OpenGL calls. Do not attempt to call X, any of the functions in <FL/fl_draw.H>, or glX directly. Do not call gl_start() or gl_finish().

If double-buffering is enabled in the window, the back and front buffers are swapped after this function is completed.

Reimplemented from Fl_Window.

Reimplemented in Fl_Glut_Window.

void Fl_Gl_Window::flush ( ) `[virtual]`

Forces the window to be drawn, this window is also made current and calls draw().

Reimplemented from Fl_Window.

void Fl_Gl_Window::hide_overlay ( )

Hides the window if it is not this window, does nothing in WIN32.

void Fl_Gl_Window::make_current ( )

The make_current() method selects the OpenGL context for the widget.

It is called automatically prior to the draw() method being called and can also be used to implement feedback and/or selection within the handle() method.

void Fl_Gl_Window::make_overlay_current ( )

The make_overlay_current() method selects the OpenGL context for the widget's overlay.

It is called automatically prior to the draw_overlay() method being called and can also be used to implement feedback and/or selection within the handle() method.

Fl_Mode Fl_Gl_Window::mode ( ) const `[inline]`

Returns the current OpenGL capabilites of the window.
Don't use this if capabilities were set through Fl_Gl_Window::mode(const int ∗a).

int Fl_Gl_Window::mode (int *a*) `[inline]`

Set or change the OpenGL capabilites of the window.
The value can be any of the following OR'd together:

- `FL_RGB` - RGB color (not indexed)
- `FL_RGB8` - RGB color with at least 8 bits of each color
- `FL_INDEX` - Indexed mode
- `FL_SINGLE` - not double buffered
- `FL_DOUBLE` - double buffered
- `FL_ACCUM` - accumulation buffer
- `FL_ALPHA` - alpha channel in color
- `FL_DEPTH` - depth buffer
- `FL_STENCIL` - stencil buffer
- `FL_MULTISAMPLE` - multisample antialiasing
- `FL_OPENGL3` - use OpenGL version 3.0 or more when running Mac OS.

FL_RGB and FL_SINGLE have a value of zero, so they are "on" unless you give FL_INDEX or FL_DOUBLE.

If the desired combination cannot be done, FLTK will try turning off FL_MULTISAMPLE. If this also fails the show() will call Fl::error() and not show the window.

You can change the mode while the window is displayed. This is most useful for turning double-buffering on and off. Under X this will cause the old X window to be destroyed and a new one to be created. If this is a top-level window this will unfortunately also cause the window to blink, raise to the top, and be de-iconized, and the xid() will change, possibly breaking other code. It is best to make the GL window a child of another window if you wish to do this!

mode() must not be called within draw() since it changes the current context.

Note

> On the **MSWindows and Unix/Linux platforms**, FLTK produces contexts for the highest OpenGL version supported by the hardware. Such contexts are also compatible with lower OpenGL versions. On the **Apple OS X platform**, it is necessary to decide whether the source code targets OpenGL versions higher or lower than 3.0. By default, FLTK creates contexts adequate for OpenGL versions 1 and 2. To get contexts for OpenGL 3.0 or higher, the `FL_OPENGL3` flag and Mac OS version 10.7 or higher are required (in that case the context is NOT compatible with OpenGL versions 1 or 2). The `FL_OPENGL3` flag has no effect on non-Apple platforms.

Version

> the `FL_OPENGL3` flag appeared in version 1.3.4

int Fl_Gl_Window::mode (const int ∗ *a*) `[inline]`

Set the OpenGL capabilites of the window using platform-specific data.

Parameters

a	zero-ending array of platform-specific attributes and attribute values

Unix/Linux platform: attributes are GLX attributes adequate for the 3rd argument of the `glXChoose-Visual()` function (e.g., `GLX_DOUBLEBUFFER`, defined by including <GL/glx.h>).

Note

What attributes are adequate here is subject to change. The preferred, stable public API is Fl_Gl_-Window::mode(int a).

MSWindows platform: this member function is of no use.

Mac OS X platform: attributes belong to the `CGLPixelFormatAttribute` enumeration (defined by including `<OpenGL/OpenGL.h>`, e.g., `kCGLPFADoubleBuffer`) and may be followed by adequate attribute values.

void Fl_Gl_Window::ortho ( )

Sets the projection so 0,0 is in the lower left of the window and each pixel is 1 unit wide/tall.

If you are drawing 2D images, your draw() method may want to call this if valid() is false.

int Fl_Gl_Window::pixel_h ( ) `[inline]`

Gives the window height in OpenGL pixels.

Generally identical with the result of the h() function, but for a window mapped to an Apple 'retina' display, and if Fl::use_high_res_GL(bool) is set to true, pixel_h() returns 2 ∗ h(). This method detects when the window has been moved between low and high resolution displays and automatically adjusts the returned value.

Version

1.3.4

int Fl_Gl_Window::pixel_w ( ) `[inline]`

Gives the window width in OpenGL pixels.

Generally identical with the result of the w() function, but for a window mapped to an Apple 'retina' display, and if Fl::use_high_res_GL(bool) is set to true, pixel_w() returns 2 ∗ w(). This method detects when the window has been moved between low and high resolution displays and automatically adjusts the returned value.

Version

1.3.4

float Fl_Gl_Window::pixels_per_unit ( ) `[inline]`

The number of pixels per FLTK unit of length for the window.

Returns 1, except for a window mapped to an Apple 'retina' display, and if Fl::use_high_res_GL(bool) is set to true, when it returns 2. This method dynamically adjusts its value when the window is moved to/from a retina display. This method is useful, e.g., to convert, in a window's handle() method, the FLTK units returned by Fl::event_x() and Fl::event_y() to the pixel units used by the OpenGL source code.

Version

1.3.4

void Fl_Gl_Window::redraw_overlay ( )

This method causes draw_overlay() to be called at a later time.

Initially the overlay is clear. If you want the window to display something in the overlay when it first appears, you must call this immediately after you show() your window.

void Fl_Gl_Window::resize (int *X*, int *Y*, int *W*, int *H*) `[virtual]`

Changes the size and position of the window.

If shown() is true, these changes are communicated to the window server (which may refuse that size and cause a further resize). If shown() is false, the size and position are used when show() is called. See Fl_Group for the effect of resizing on the child widgets.

You can also call the Fl_Widget methods size(x,y) and position(w,h), which are inline wrappers for this virtual function.

A top-level window can not force, but merely suggest a position and size to the operating system. The window manager may not be willing or able to display a window at the desired position or with the given dimensions. It is up to the application developer to verify window parameters after the resize request.

Reimplemented from Fl_Window.

void Fl_Gl_Window::show ( ) `[virtual]`

Puts the window on the screen.

Usually (on X) this has the side effect of opening the display.

If the window is already shown then it is restored and raised to the top. This is really convenient because your program can call show() at any time, even if the window is already up. It also means that show() serves the purpose of raise() in other toolkits.

Fl_Window::show(int argc, char **argv) is used for top-level windows and allows standard arguments to be parsed from the command-line.

Note

For some obscure reasons Fl_Window::show() resets the current group by calling Fl_Group::current(0). The comments in the code say "get rid of very common user bug: forgot end()". Although this is true it may have unwanted side effects if you show() an unrelated window (maybe for an error message or warning) while building a window or any other group widget.

Todo Check if we can remove resetting the current group in a later FLTK version (after 1.3.x). This may break "already broken" programs though if they rely on this "feature".

See Also

Fl_Window::show(int argc, char **argv)

Reimplemented from Fl_Window.

void Fl_Gl_Window::swap_buffers ( )

The swap_buffers() method swaps the back and front buffers.

It is called automatically after the draw() method is called.

char Fl_Gl_Window::valid ( ) const `[inline]`

Is turned off when FLTK creates a new context for this window or when the window resizes, and is turned on *after* draw() is called.

You can use this inside your draw() method to avoid unnecessarily initializing the OpenGL context. Just do this:

```
void mywindow::draw() {
 if (!valid()) {
   glViewport(0,0,pixel_w(),pixel_h());
   glFrustum(...);
   ...other initialization...
 }
 if (!context_valid()) {
   ...load textures, etc. ...
 }
 ... draw your geometry here ...
}
```

You can turn valid() on by calling valid(1). You should only do this after fixing the transformation inside a draw() or after make_current(). This is done automatically after draw() returns.

The documentation for this class was generated from the following files:

- Fl_Gl_Window.H
- Fl_Gl_Overlay.cxx
- Fl_Gl_Window.cxx

31.47 Fl_Glut_Bitmap_Font Struct Reference

fltk glut font/size attributes used in the glutXXX functions

`#include <glut.H>`

Public Attributes

- Fl_Font **font**
- Fl_Fontsize **size**

31.47.1 Detailed Description

fltk glut font/size attributes used in the glutXXX functions

The documentation for this struct was generated from the following file:

- glut.H

31.48 Fl_Glut_StrokeChar Struct Reference

Public Attributes

- int **Number**
- GLfloat **Right**
- const Fl_Glut_StrokeStrip ∗ **Strips**

The documentation for this struct was generated from the following file:

- glut.H

31.49 Fl_Glut_StrokeFont Struct Reference

Public Attributes

- const Fl_Glut_StrokeChar ∗∗ **Characters**
- GLfloat **Height**
- char ∗ **Name**
- int **Quantity**

The documentation for this struct was generated from the following file:

- glut.H

31.50 Fl_Glut_StrokeStrip Struct Reference

Public Attributes

- int **Number**
- const Fl_Glut_StrokeVertex ∗ **Vertices**

The documentation for this struct was generated from the following file:

- glut.H

31.51 Fl_Glut_StrokeVertex Struct Reference

Public Attributes

- GLfloat **X**
- GLfloat **Y**

The documentation for this struct was generated from the following file:

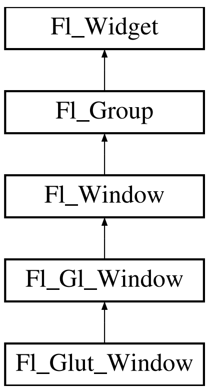
- glut.H

31.52 Fl_Glut_Window Class Reference

GLUT is emulated using this window class and these static variables (plus several more static variables hidden in glut_compatability.cxx):

```
#include <glut.H>
```

Inheritance diagram for Fl_Glut_Window:



Public Member Functions

- Fl_Glut_Window (int w, int h, const char ∗)

  Creates a glut window, registers to the glut windows list.
- Fl_Glut_Window (int x, int y, int w, int h, const char ∗)

  Creates a glut window, registers to the glut windows list.

- void **make_current** ()
- ∼Fl_Glut_Window ()

  Destroys the glut window, first unregister it from the glut windows list.

Public Attributes

- void(∗ **display**)()
- void(∗ **entry**)(int)
- void(∗ **keyboard**)(uchar, int x, int y)
- int **menu** [3]
- void(∗ **motion**)(int x, int y)
- void(∗ **mouse**)(int b, int state, int x, int y)
- int **number**
- void(∗ **overlaydisplay**)()
- void(∗ **passivemotion**)(int x, int y)
- void(∗ **reshape**)(int w, int h)
- void(∗ **special**)(int, int x, int y)
- void(∗ **visibility**)(int)

Protected Member Functions

- void draw ()

  Draws the Fl_Gl_Window.
- void draw_overlay ()

  You must implement this virtual function if you want to draw into the overlay.
- int handle (int)

  Handle some FLTK events as needed.

Additional Inherited Members

31.52.1 Detailed Description

GLUT is emulated using this window class and these static variables (plus several more static variables hidden in glut_compatability.cxx):

31.52.2 Constructor & Destructor Documentation

Fl_Glut_Window::Fl_Glut_Window (int *W*, int *H*, const char ∗ *t*)

Creates a glut window, registers to the glut windows list.

Fl_Glut_Window::Fl_Glut_Window (int *X*, int *Y*, int *W*, int *H*, const char ∗ *t*)

Creates a glut window, registers to the glut windows list.

31.52.3 Member Function Documentation

void Fl_Glut_Window::draw (void) [protected], [virtual]

Draws the Fl_Gl_Window.

You ***must*** *subclass* Fl_Gl_Window and provide an implementation for draw().

You ***must*** *override* the draw() method.

You may also provide an implementation of draw_overlay() if you want to draw into the overlay planes. You can avoid reinitializing the viewport and lights and other things by checking valid() at the start of draw() and only doing the initialization if it is false.

The draw() method can *only* use OpenGL calls. Do not attempt to call X, any of the functions in <FL/fl_draw.H>, or glX directly. Do not call gl_start() or gl_finish().

If double-buffering is enabled in the window, the back and front buffers are swapped after this function is completed.

Reimplemented from Fl_Gl_Window.

void Fl_Glut_Window::draw_overlay ( ) [protected], [virtual]

You must implement this virtual function if you want to draw into the overlay.

The overlay is cleared before this is called. You should draw anything that is not clear using OpenGL. You must use gl_color(i) to choose colors (it allocates them from the colormap using system-specific calls), and remember that you are in an indexed OpenGL mode and drawing anything other than flat-shaded will probably not work.

Both this function and Fl_Gl_Window::draw() should check Fl_Gl_Window::valid() and set the same transformation. If you don't your code may not work on other systems. Depending on the OS, and on whether overlays are real or simulated, the OpenGL context may be the same or different between the overlay and main window.

Reimplemented from Fl_Gl_Window.

The documentation for this class was generated from the following files:

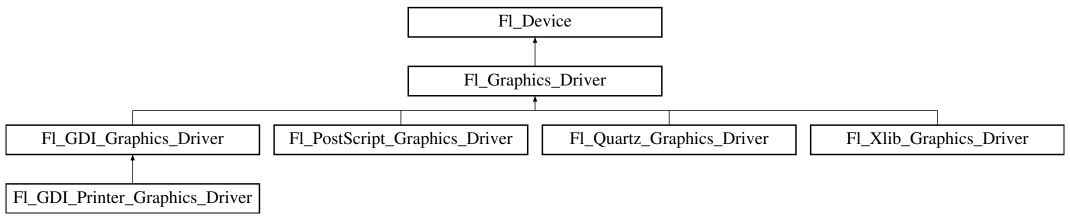
- glut.H
- glut_compatability.cxx

31.53 Fl_Graphics_Driver Class Reference

A virtual class subclassed for each graphics driver FLTK uses.

`#include <Fl_Device.H>`

Inheritance diagram for Fl_Graphics_Driver:


Fl_Device
Fl_Graphics_Driver
Fl_GDI_Graphics_Driver
Fl_PostScript_Graphics_Driver
Fl_Quartz_Graphics_Driver
Fl_Xlib_Graphics_Driver
Fl_GDI_Printer_Graphics_Driver

Classes

- struct matrix

  A 2D coordinate transformation matrix.

Public Member Functions

- virtual const char ∗ class_name ()

  Returns the name of the class of this object.
- Fl_Color color ()

  see fl_color(void).
- virtual int descent ()

  see fl_descent().
- virtual int draw_scaled (Fl_Image ∗img, int X, int Y, int W, int H)

  Draws an Fl_Image scaled to width `W` & height `H` with top-left corner at X,Y.
- virtual void font (Fl_Font face, Fl_Fontsize fsize)

  see fl_font(Fl_Font face, Fl_Fontsize size).
- Fl_Font font ()

  see fl_font(void).
- Fl_Font_Descriptor ∗ font_descriptor ()

  Returns a pointer to the current Fl_Font_Descriptor for the graphics driver.
- void font_descriptor (Fl_Font_Descriptor ∗d)

  Sets the current Fl_Font_Descriptor for the graphics driver.
- virtual int height ()

  see fl_height().
- Fl_Fontsize size ()

  see fl_size().
- virtual void text_extents (const char ∗, int n, int &dx, int &dy, int &w, int &h)

  see fl_text_extents(const char∗, int n, int& dx, int& dy, int& w, int& h).
- virtual double width (const char ∗str, int n)

  see fl_width(const char ∗str, int n).
- virtual double width (unsigned int c)

  see fl_width(unsigned int n).
- virtual ∼Fl_Graphics_Driver ()

  The destructor.

Static Public Attributes

- static const char ∗ **class_id** = "Fl_Graphics_Driver"

Protected Member Functions

- virtual void arc (double x, double y, double r, double start, double end)

  see fl_arc(double x, double y, double r, double start, double end).
- virtual void arc (int x, int y, int w, int h, double a1, double a2)

  see fl_arc(int x, int y, int w, int h, double a1, double a2).
- virtual void begin_complex_polygon ()

  see fl_begin_complex_polygon().
- virtual void begin_line ()

  see fl_begin_line().
- virtual void begin_loop ()

  see fl_begin_loop().
- virtual void begin_points ()

see fl_begin_points().

- virtual void begin_polygon ()

  see fl_begin_polygon().
- virtual void circle (double x, double y, double r)

  see fl_circle(double x, double y, double r).
- virtual int clip_box (int x, int y, int w, int h, int &X, int &Y, int &W, int &H)

  see fl_clip_box(int x, int y, int w, int h, int &X, int &Y, int &W, int &H).
- Fl_Region clip_region ()

  see fl_clip_region().
- void clip_region (Fl_Region r)

  see fl_clip_region(Fl_Region r).
- virtual void color (Fl_Color c)

  see fl_color(Fl_Color c).
- virtual void color (uchar r, uchar g, uchar b)

  see fl_color(uchar r, uchar g, uchar b).
- virtual void copy_offscreen (int x, int y, int w, int h, Fl_Offscreen pixmap, int srcx, int srcy)

  see fl_copy_offscreen()
- virtual void curve (double X0, double Y0, double X1, double Y1, double X2, double Y2, double X3, double Y3)

  see fl_curve(double X0, double Y0, double X1, double Y1, double X2, double Y2, double X3, double Y3).
- virtual void draw (const char ∗str, int n, int x, int y)

  see fl_draw(const char ∗str, int n, int x, int y).
- virtual void draw (int angle, const char ∗str, int n, int x, int y)

  see fl_draw(int angle, const char ∗str, int n, int x, int y).
- virtual void draw (Fl_RGB_Image ∗rgb, int XP, int YP, int WP, int HP, int cx, int cy)

  Draws an Fl_RGB_Image object to the device.
- virtual void draw (Fl_Pixmap ∗pxm, int XP, int YP, int WP, int HP, int cx, int cy)

  Draws an Fl_Pixmap object to the device.
- virtual void draw (Fl_Bitmap ∗bm, int XP, int YP, int WP, int HP, int cx, int cy)

  Draws an Fl_Bitmap object to the device.
- virtual void draw_image (const uchar ∗buf, int X, int Y, int W, int H, int D=3, int L=0)

  see fl_draw_image(const uchar∗ buf, int X,int Y,int W,int H, int D, int L).
- virtual void draw_image (Fl_Draw_Image_Cb cb, void ∗data, int X, int Y, int W, int H, int D=3)

  see fl_draw_image(Fl_Draw_Image_Cb cb, void∗ data, int X,int Y,int W,int H, int D).
- virtual void draw_image_mono (const uchar ∗buf, int X, int Y, int W, int H, int D=1, int L=0)

  see fl_draw_image_mono(const uchar∗ buf, int X,int Y,int W,int H, int D, int L).
- virtual void draw_image_mono (Fl_Draw_Image_Cb cb, void ∗data, int X, int Y, int W, int H, int D=1)

  see fl_draw_image_mono(Fl_Draw_Image_Cb cb, void∗ data, int X,int Y,int W,int H, int D).
- virtual void end_complex_polygon ()

  see fl_end_complex_polygon().
- virtual void end_line ()

  see fl_end_line().
- virtual void end_loop ()

  see fl_end_loop().
- virtual void end_points ()

  see fl_end_points().
- virtual void end_polygon ()

see fl_end_polygon().
- Fl_Graphics_Driver ()

  The constructor.
- virtual void gap ()

  see fl_gap().
- virtual void line (int x, int y, int x1, int y1)

  see fl_line(int x, int y, int x1, int y1).
- virtual void line (int x, int y, int x1, int y1, int x2, int y2)

  see fl_line(int x, int y, int x1, int y1, int x2, int y2).
- virtual void line_style (int style, int width=0, char ∗dashes=0)

  see fl_line_style(int style, int width, char∗ dashes).
- virtual void loop (int x0, int y0, int x1, int y1, int x2, int y2)

  see fl_loop(int x0, int y0, int x1, int y1, int x2, int y2).
- virtual void loop (int x0, int y0, int x1, int y1, int x2, int y2, int x3, int y3)

  see fl_loop(int x0, int y0, int x1, int y1, int x2, int y2, int x3, int y3).
- void mult_matrix (double a, double b, double c, double d, double x, double y)

  see fl_mult_matrix(double a, double b, double c, double d, double x, double y).
- virtual int not_clipped (int x, int y, int w, int h)

  see fl_not_clipped(int x, int y, int w, int h).
- virtual void pie (int x, int y, int w, int h, double a1, double a2)

  see fl_pie(int x, int y, int w, int h, double a1, double a2).
- virtual void point (int x, int y)

  see fl_point(int x, int y).
- virtual void polygon (int x0, int y0, int x1, int y1, int x2, int y2)

  see fl_polygon(int x0, int y0, int x1, int y1, int x2, int y2).
- virtual void polygon (int x0, int y0, int x1, int y1, int x2, int y2, int x3, int y3)

  see fl_polygon(int x0, int y0, int x1, int y1, int x2, int y2, int x3, int y3).
- virtual void pop_clip ()

  see fl_pop_clip().
- void pop_matrix ()

  see fl_pop_matrix().
- virtual void push_clip (int x, int y, int w, int h)

  see fl_push_clip(int x, int y, int w, int h).
- void push_matrix ()

  see fl_push_matrix().
- virtual void push_no_clip ()

  see fl_push_no_clip().
- virtual void rect (int x, int y, int w, int h)

  see fl_rect(int x, int y, int w, int h).
- virtual void rectf (int x, int y, int w, int h)

  see fl_rectf(int x, int y, int w, int h).
- void restore_clip ()

  see fl_restore_clip().
- void rotate (double d)

  see fl_rotate(double d).
- virtual void rtl_draw (const char ∗str, int n, int x, int y)

see fl_rtl_draw(const char ∗str, int n, int x, int y).

- void scale (double x, double y)

  see fl_scale(double x, double y).
- void scale (double x)

  see fl_scale(double x).
- double transform_dx (double x, double y)

  see fl_transform_dx(double x, double y).
- double transform_dy (double x, double y)

  see fl_transform_dy(double x, double y).
- double transform_x (double x, double y)

  see fl_transform_x(double x, double y).
- double transform_y (double x, double y)

  see fl_transform_y(double x, double y).
- virtual void transformed_vertex (double xf, double yf)

  see fl_transformed_vertex(double xf, double yf).
- void translate (double x, double y)

  see fl_translate(double x, double y).
- virtual void vertex (double x, double y)

  see fl_vertex(double x, double y).
- virtual void xyline (int x, int y, int x1)

  see fl_xyline(int x, int y, int x1).
- virtual void xyline (int x, int y, int x1, int y2)

  see fl_xyline(int x, int y, int x1, int y2).
- virtual void xyline (int x, int y, int x1, int y2, int x3)

  see fl_xyline(int x, int y, int x1, int y2, int x3).
- virtual void yxline (int x, int y, int y1)

  see fl_yxline(int x, int y, int y1).
- virtual void yxline (int x, int y, int y1, int x2)

  see fl_yxline(int x, int y, int y1, int x2).
- virtual void yxline (int x, int y, int y1, int x2, int y3)

  see fl_yxline(int x, int y, int y1, int x2, int y3).

Protected Attributes

- matrix ∗ fl_matrix

  Points to the current coordinate transformation matrix.

Friends

- void fl_arc (double x, double y, double r, double start, double end)

  Adds a series of points to the current path on the arc of a circle.
- void fl_arc (int x, int y, int w, int h, double a1, double a2)

  Draw ellipse sections using integer coordinates.
- void fl_begin_complex_polygon ()

  Starts drawing a complex filled polygon.
- void fl_begin_line ()

  Starts drawing a list of lines.
- void fl_begin_loop ()

Starts drawing a closed sequence of lines.
- void fl_begin_points ()

  Starts drawing a list of points.
- void fl_begin_polygon ()

  Starts drawing a convex filled polygon.
- class **Fl_Bitmap**
- void fl_circle (double x, double y, double r)

  fl_circle() is equivalent to fl_arc(x,y,r,0,360), but may be faster.
- int fl_clip_box (int x, int y, int w, int h, int &X, int &Y, int &W, int &H)

  Intersects the rectangle with the current clip region and returns the bounding box of the result.
- Fl_Region fl_clip_region ()

  Returns the current clipping region.
- void fl_clip_region (Fl_Region r)

  Replaces the top of the clipping stack with a clipping region of any shape.
- void fl_color (Fl_Color c)

  Sets the color for all subsequent drawing operations.
- void fl_color (uchar r, uchar g, uchar b)

  Sets the color for all subsequent drawing operations.
- FL_EXPORT void fl_copy_offscreen (int x, int y, int w, int h, Fl_Offscreen pixmap, int srcx, int srcy)

  Copy a rectangular area of the given offscreen buffer into the current drawing destination.
- void fl_curve (double X0, double Y0, double X1, double Y1, double X2, double Y2, double X3, double Y3)

  Adds a series of points on a Bezier curve to the path.
- void fl_draw (const char ∗str, int n, int x, int y)

  Draws starting at the given `x`, `y` location a UTF-8 string of length `n` bytes.
- void fl_draw (int angle, const char ∗str, int n, int x, int y)

  Draws at the given `x`, `y` location a UTF-8 string of length `n` bytes rotating `angle` degrees counter-clockwise.
- void fl_draw_image (const uchar ∗buf, int X, int Y, int W, int H, int D, int L)

  Draws an 8-bit per color RGB or luminance image.
- void fl_draw_image (Fl_Draw_Image_Cb cb, void ∗data, int X, int Y, int W, int H, int D)

  Draws an image using a callback function to generate image data.
- void fl_draw_image_mono (const uchar ∗buf, int X, int Y, int W, int H, int D, int L)

  Draws a gray-scale (1 channel) image.
- FL_EXPORT void fl_draw_image_mono (Fl_Draw_Image_Cb cb, void ∗data, int X, int Y, int W, int H, int D)

  Draws a gray-scale image using a callback function to generate image data.
- void fl_end_complex_polygon ()

  Ends complex filled polygon, and draws.
- void fl_end_line ()

  Ends list of lines, and draws.
- void fl_end_loop ()

  Ends closed sequence of lines, and draws.
- void fl_end_points ()

  Ends list of points, and draws.
- void fl_end_polygon ()

  Ends convex filled polygon, and draws.

- void fl_font (Fl_Font face, Fl_Fontsize size)

    Sets the current font, which is then used in various drawing routines.
- void fl_gap ()

    Call fl_gap() to separate loops of the path.
- void fl_line (int x, int y, int x1, int y1)

    Draws a line from (x,y) to (x1,y1)
- void fl_line (int x, int y, int x1, int y1, int x2, int y2)

    Draws a line from (x,y) to (x1,y1) and another from (x1,y1) to (x2,y2)
- void fl_line_style (int style, int width, char ∗dashes)

    Sets how to draw lines (the "pen").
- void fl_loop (int x0, int y0, int x1, int y1, int x2, int y2)

    Outlines a 3-sided polygon with lines.
- void fl_loop (int x0, int y0, int x1, int y1, int x2, int y2, int x3, int y3)

    Outlines a 4-sided polygon with lines.
- void fl_mult_matrix (double a, double b, double c, double d, double x, double y)

    Concatenates another transformation onto the current one.
- int fl_not_clipped (int x, int y, int w, int h)

    Does the rectangle intersect the current clip region?
- void fl_pie (int x, int y, int w, int h, double a1, double a2)

    Draw filled ellipse sections using integer coordinates.
- class **Fl_Pixmap**
- void fl_point (int x, int y)

    Draws a single pixel at the given coordinates.
- void fl_polygon (int x0, int y0, int x1, int y1, int x2, int y2)

    Fills a 3-sided polygon.
- void fl_polygon (int x0, int y0, int x1, int y1, int x2, int y2, int x3, int y3)

    Fills a 4-sided polygon.
- void fl_pop_clip ()

    Restores the previous clip region.
- void fl_pop_matrix ()

    Restores the current transformation matrix from the stack.
- void fl_push_clip (int x, int y, int w, int h)

    Intersects the current clip region with a rectangle and pushes this new region onto the stack.
- void fl_push_matrix ()

    Saves the current transformation matrix on the stack.
- void fl_push_no_clip ()

    Pushes an empty clip region onto the stack so nothing will be clipped.
- void fl_rect (int x, int y, int w, int h)

    Draws a 1-pixel border inside the given bounding box.
- void fl_rectf (int x, int y, int w, int h)

    Colors with current color a rectangle that exactly fills the given bounding box.
- void fl_restore_clip ()

    Undoes any clobbering of clip done by your program.
- class **Fl_RGB_Image**
- void fl_rotate (double d)

    Concatenates rotation transformation onto the current one.
- void fl_rtl_draw (const char ∗str, int n, int x, int y)

Draws a UTF-8 string of length `n` *bytes right to left starting at the given* `x`, `y` *location.*

- void fl_scale (double x, double y)

  Concatenates scaling transformation onto the current one.
- void fl_scale (double x)

  Concatenates scaling transformation onto the current one.
- double fl_transform_dx (double x, double y)

  Transforms distance using current transformation matrix.
- double fl_transform_dy (double x, double y)

  Transforms distance using current transformation matrix.
- double fl_transform_x (double x, double y)

  Transforms coordinate using the current transformation matrix.
- double fl_transform_y (double x, double y)

  Transforms coordinate using the current transformation matrix.
- void fl_transformed_vertex (double xf, double yf)

  Adds coordinate pair to the vertex list without further transformations.
- void fl_translate (double x, double y)

  Concatenates translation transformation onto the current one.
- void fl_vertex (double x, double y)

  Adds a single vertex to the current path.
- void fl_xyline (int x, int y, int x1)

  Draws a horizontal line from (x,y) to (x1,y)
- void fl_xyline (int x, int y, int x1, int y2)

  Draws a horizontal line from (x,y) to (x1,y), then vertical from (x1,y) to (x1,y2)
- void fl_xyline (int x, int y, int x1, int y2, int x3)

  Draws a horizontal line from (x,y) to (x1,y), then a vertical from (x1,y) to (x1,y2) and then another horizontal from (x1,y2) to (x3,y2)
- void fl_yxline (int x, int y, int y1)

  Draws a vertical line from (x,y) to (x,y1)
- void fl_yxline (int x, int y, int y1, int x2)

  Draws a vertical line from (x,y) to (x,y1), then a horizontal from (x,y1) to (x2,y1)
- void fl_yxline (int x, int y, int y1, int x2, int y3)

  Draws a vertical line from (x,y) to (x,y1) then a horizontal from (x,y1) to (x2,y1), then another vertical from (x2,y1) to (x2,y3)
- FL_EXPORT void gl_start ()

  Creates an OpenGL context.

31.53.1 Detailed Description

A virtual class subclassed for each graphics driver FLTK uses.

Typically, FLTK applications do not use directly objects from this class. Rather, they perform drawing operations (e.g., fl_rectf()) that operate on the current drawing surface (see Fl_Surface_Device). Drawing operations are functionally presented in Drawing Things in FLTK and as function lists in the Drawing functions and Color & Font functions modules. The fl_graphics_driver global variable gives at any time the graphics driver used by all drawing operations. Its value changes when drawing operations are directed to another drawing surface by Fl_Surface_Device::set_current().

`The` Fl_Graphics_Driver class is of interest if one wants to perform new kinds of drawing operations. An example would be to draw to a PDF file. This would involve creating a new Fl_Graphics_Driver derived class. This new class should implement all virtual methods of the Fl_Graphics_Driver class to support all FLTK drawing functions.

31.53.2 Constructor & Destructor Documentation

Fl_Graphics_Driver::Fl_Graphics_Driver ( ) `[protected]`

The constructor.

31.53.3 Member Function Documentation

void Fl_Graphics_Driver::arc (double *x*, double *y*, double *r*, double *start*, double *end*) `[protected]`, `[virtual]`

see fl_arc(double x, double y, double r, double start, double end).
Reimplemented in Fl_PostScript_Graphics_Driver.

void Fl_Graphics_Driver::arc (int *x*, int *y*, int *w*, int *h*, double *a1*, double *a2*) `[protected]`, `[virtual]`

see fl_arc(int x, int y, int w, int h, double a1, double a2).
Reimplemented in Fl_PostScript_Graphics_Driver.

void Fl_Graphics_Driver::begin_complex_polygon ( ) `[protected]`, `[virtual]`

see fl_begin_complex_polygon().
Reimplemented in Fl_PostScript_Graphics_Driver.

void Fl_Graphics_Driver::begin_line ( ) `[protected]`, `[virtual]`

see fl_begin_line().
Reimplemented in Fl_PostScript_Graphics_Driver.

void Fl_Graphics_Driver::begin_loop ( ) `[protected]`, `[virtual]`

see fl_begin_loop().
Reimplemented in Fl_PostScript_Graphics_Driver.

void Fl_Graphics_Driver::begin_points ( ) `[protected]`, `[virtual]`

see fl_begin_points().
Reimplemented in Fl_PostScript_Graphics_Driver.

void Fl_Graphics_Driver::begin_polygon ( ) `[protected]`, `[virtual]`

see fl_begin_polygon().
Reimplemented in Fl_PostScript_Graphics_Driver.

void Fl_Graphics_Driver::circle (double *x*, double *y*, double *r*) `[protected]`, `[virtual]`

see fl_circle(double x, double y, double r).
Reimplemented in Fl_PostScript_Graphics_Driver.

virtual const char∗ Fl_Graphics_Driver::class_name ( ) `[inline], [virtual]`

Returns the name of the class of this object.

Use of the class_name() function is discouraged because it will be removed from future FLTK versions.

The class of an instance of an Fl_Device subclass can be checked with code such as:

```
if ( instance->class_name() == Fl_Printer::class_id ) { ... }
```

Reimplemented from Fl_Device.

Reimplemented in Fl_Xlib_Graphics_Driver, Fl_GDI_Printer_Graphics_Driver, Fl_GDI_Graphics_Driver, Fl_Quartz_Graphics_Driver, and Fl_PostScript_Graphics_Driver.

int Fl_Graphics_Driver::clip_box (int *x*, int *y*, int *w*, int *h*, int & *X*, int & *Y*, int & *W*, int & *H*) `[protected], [virtual]`

see fl_clip_box(int x, int y, int w, int h, int &X, int &Y, int &W, int &H).

Reimplemented in Fl_PostScript_Graphics_Driver.

Fl_Region Fl_Graphics_Driver::clip_region ( ) `[protected]`

see fl_clip_region().

void Fl_Graphics_Driver::clip_region (Fl_Region *r*) `[protected]`

see fl_clip_region(Fl_Region r).

virtual void Fl_Graphics_Driver::color (Fl_Color *c*) `[inline], [protected], [virtual]`

see fl_color(Fl_Color c).

Reimplemented in Fl_Xlib_Graphics_Driver, Fl_GDI_Graphics_Driver, Fl_Quartz_Graphics_Driver, and Fl_PostScript_Graphics_Driver.

virtual void Fl_Graphics_Driver::color (uchar *r*, uchar *g*, uchar *b*) `[inline], [protected], [virtual]`

see fl_color(uchar r, uchar g, uchar b).

Reimplemented in Fl_Xlib_Graphics_Driver, Fl_GDI_Graphics_Driver, Fl_Quartz_Graphics_Driver, and Fl_PostScript_Graphics_Driver.

Fl_Color Fl_Graphics_Driver::color ( ) `[inline]`

see fl_color(void).

void Fl_Graphics_Driver::curve (double *X0*, double *Y0*, double *X1*, double *Y1*, double *X2*, double *Y2*, double *X3*, double *Y3*) `[protected], [virtual]`

see fl_curve(double X0, double Y0, double X1, double Y1, double X2, double Y2, double X3, double Y3).

Reimplemented in Fl_PostScript_Graphics_Driver.

virtual int Fl_Graphics_Driver::descent ( ) `[inline], [virtual]`

see fl_descent().

Reimplemented in Fl_Xlib_Graphics_Driver, Fl_GDI_Graphics_Driver, Fl_Quartz_Graphics_Driver, and Fl_PostScript_Graphics_Driver.

virtual void Fl_Graphics_Driver::draw (const char * *str*, int *n*, int *x*, int *y*) `[inline]`, `[protected]`, `[virtual]`

see fl_draw(const char *str, int n, int x, int y).

Reimplemented in Fl_Xlib_Graphics_Driver, Fl_GDI_Graphics_Driver, Fl_Quartz_Graphics_Driver, and Fl_PostScript_Graphics_Driver.

virtual void Fl_Graphics_Driver::draw (int *angle*, const char * *str*, int *n*, int *x*, int *y*) `[inline]`, `[protected]`, `[virtual]`

see fl_draw(int angle, const char *str, int n, int x, int y).

Reimplemented in Fl_Xlib_Graphics_Driver, Fl_GDI_Graphics_Driver, Fl_Quartz_Graphics_Driver, and Fl_PostScript_Graphics_Driver.

virtual void Fl_Graphics_Driver::draw (Fl_RGB_Image * *rgb*, int *XP*, int *YP*, int *WP*, int *HP*, int *cx*, int *cy*) `[inline]`, `[protected]`, `[virtual]`

Draws an Fl_RGB_Image object to the device.

Specifies a bounding box for the image, with the origin (upper left-hand corner) of the image offset by the cx and cy arguments.

Reimplemented in Fl_Xlib_Graphics_Driver, Fl_GDI_Graphics_Driver, Fl_Quartz_Graphics_Driver, and Fl_PostScript_Graphics_Driver.

virtual void Fl_Graphics_Driver::draw (Fl_Pixmap * *pxm*, int *XP*, int *YP*, int *WP*, int *HP*, int *cx*, int *cy*) `[inline]`, `[protected]`, `[virtual]`

Draws an Fl_Pixmap object to the device.

Specifies a bounding box for the image, with the origin (upper left-hand corner) of the image offset by the cx and cy arguments.

Reimplemented in Fl_Xlib_Graphics_Driver, Fl_GDI_Printer_Graphics_Driver, Fl_GDI_Graphics_Driver, Fl_Quartz_Graphics_Driver, and Fl_PostScript_Graphics_Driver.

virtual void Fl_Graphics_Driver::draw (Fl_Bitmap * *bm*, int *XP*, int *YP*, int *WP*, int *HP*, int *cx*, int *cy*) `[inline]`, `[protected]`, `[virtual]`

Draws an Fl_Bitmap object to the device.

Specifies a bounding box for the image, with the origin (upper left-hand corner) of the image offset by the cx and cy arguments.

Reimplemented in Fl_Xlib_Graphics_Driver, Fl_GDI_Printer_Graphics_Driver, Fl_GDI_Graphics_Driver, Fl_Quartz_Graphics_Driver, and Fl_PostScript_Graphics_Driver.

virtual void Fl_Graphics_Driver::draw_image (const uchar * *buf*, int *X*, int *Y*, int *W*, int *H*, int *D* = *3*, int *L* = *0*) `[inline]`, `[protected]`, `[virtual]`

see fl_draw_image(const uchar* buf, int X,int Y,int W,int H, int D, int L).

Reimplemented in Fl_Xlib_Graphics_Driver, Fl_GDI_Graphics_Driver, Fl_Quartz_Graphics_Driver, and Fl_PostScript_Graphics_Driver.

virtual void Fl_Graphics_Driver::draw_image (Fl_Draw_Image_Cb *cb*, void * *data*, int *X*, int *Y*, int *W*, int *H*, int *D* = *3*) `[inline]`, `[protected]`, `[virtual]`

see fl_draw_image(Fl_Draw_Image_Cb cb, void* data, int X,int Y,int W,int H, int D).

Reimplemented in Fl_Xlib_Graphics_Driver, Fl_GDI_Graphics_Driver, Fl_Quartz_Graphics_Driver, and Fl_PostScript_Graphics_Driver.

virtual void Fl_Graphics_Driver::draw_image_mono (const uchar ∗ *buf*, int *X*, int *Y*, int *W*, int *H*, int *D* = `1`, int *L* = `0`) `[inline]`, `[protected]`, `[virtual]`

see fl_draw_image_mono(const uchar∗ buf, int X,int Y,int W,int H, int D, int L).
Reimplemented in Fl_Xlib_Graphics_Driver, Fl_GDI_Graphics_Driver, Fl_Quartz_Graphics_Driver, and Fl_PostScript_Graphics_Driver.

virtual void Fl_Graphics_Driver::draw_image_mono (Fl_Draw_Image_Cb *cb*, void ∗ *data*, int *X*, int *Y*, int *W*, int *H*, int *D* = `1`) `[inline]`, `[protected]`, `[virtual]`

see fl_draw_image_mono(Fl_Draw_Image_Cb cb, void∗ data, int X,int Y,int W,int H, int D).
Reimplemented in Fl_Xlib_Graphics_Driver, Fl_GDI_Graphics_Driver, Fl_Quartz_Graphics_Driver, and Fl_PostScript_Graphics_Driver.

int Fl_Graphics_Driver::draw_scaled (Fl_Image ∗ *img*, int *X*, int *Y*, int *W*, int *H*) `[virtual]`

Draws an Fl_Image scaled to width `W` & height `H` with top-left corner at *X*,Y.

Returns

zero when the graphics driver doesn't implement scaled drawing, non-zero if it does implement it.

Reimplemented in Fl_GDI_Printer_Graphics_Driver, Fl_Quartz_Graphics_Driver, and Fl_PostScript_Graphics_Driver.

void Fl_Graphics_Driver::end_complex_polygon ( ) `[protected]`, `[virtual]`

see fl_end_complex_polygon().
Reimplemented in Fl_PostScript_Graphics_Driver.

void Fl_Graphics_Driver::end_line ( ) `[protected]`, `[virtual]`

see fl_end_line().
Reimplemented in Fl_PostScript_Graphics_Driver.

void Fl_Graphics_Driver::end_loop ( ) `[protected]`, `[virtual]`

see fl_end_loop().
Reimplemented in Fl_PostScript_Graphics_Driver.

void Fl_Graphics_Driver::end_points ( ) `[protected]`, `[virtual]`

see fl_end_points().
Reimplemented in Fl_PostScript_Graphics_Driver.

void Fl_Graphics_Driver::end_polygon ( ) `[protected]`, `[virtual]`

see fl_end_polygon().
Reimplemented in Fl_PostScript_Graphics_Driver.

virtual void Fl_Graphics_Driver::font (Fl_Font *face*, Fl_Fontsize *fsize*) `[inline]`, `[virtual]`

see fl_font(Fl_Font face, Fl_Fontsize size).
Reimplemented in Fl_Xlib_Graphics_Driver, Fl_GDI_Graphics_Driver, Fl_Quartz_Graphics_Driver, and Fl_PostScript_Graphics_Driver.

Fl_Font Fl_Graphics_Driver::font ( ) `[inline]`

see fl_font(void).

void Fl_Graphics_Driver::gap ( ) `[protected]`, `[virtual]`

see fl_gap().
Reimplemented in Fl_PostScript_Graphics_Driver.

virtual int Fl_Graphics_Driver::height ( ) `[inline]`, `[virtual]`

see fl_height().
Reimplemented in Fl_Xlib_Graphics_Driver, Fl_GDI_Graphics_Driver, Fl_Quartz_Graphics_Driver, and Fl_PostScript_Graphics_Driver.

void Fl_Graphics_Driver::line (int *x*, int *y*, int *x1*, int *y1*) `[protected]`, `[virtual]`

see fl_line(int x, int y, int x1, int y1).
Reimplemented in Fl_PostScript_Graphics_Driver.

void Fl_Graphics_Driver::line (int *x*, int *y*, int *x1*, int *y1*, int *x2*, int *y2*) `[protected]`, `[virtual]`

see fl_line(int x, int y, int x1, int y1, int x2, int y2).
Reimplemented in Fl_PostScript_Graphics_Driver.

void Fl_Graphics_Driver::line_style (int *style*, int *width* = *0*, char ∗ *dashes* = *0*) `[protected]`, `[virtual]`

see fl_line_style(int style, int width, char∗ dashes).
Reimplemented in Fl_PostScript_Graphics_Driver.

void Fl_Graphics_Driver::loop (int *x0*, int *y0*, int *x1*, int *y1*, int *x2*, int *y2*) `[protected]`, `[virtual]`

see fl_loop(int x0, int y0, int x1, int y1, int x2, int y2).
Reimplemented in Fl_PostScript_Graphics_Driver.

void Fl_Graphics_Driver::loop (int *x0*, int *y0*, int *x1*, int *y1*, int *x2*, int *y2*, int *x3*, int *y3*) `[protected]`, `[virtual]`

see fl_loop(int x0, int y0, int x1, int y1, int x2, int y2, int x3, int y3).
Reimplemented in Fl_PostScript_Graphics_Driver.

void Fl_Graphics_Driver::mult_matrix (double *a*, double *b*, double *c*, double *d*, double *x*, double *y*) `[protected]`

see fl_mult_matrix(double a, double b, double c, double d, double x, double y).

int Fl_Graphics_Driver::not_clipped (int *x*, int *y*, int *w*, int *h*) `[protected]`, `[virtual]`

see fl_not_clipped(int x, int y, int w, int h).
Reimplemented in Fl_PostScript_Graphics_Driver.

void Fl_Graphics_Driver::pie (int *x,* int *y,* int *w,* int *h,* double *a1,* double *a2*) `[protected]`, `[virtual]`

see fl_pie(int x, int y, int w, int h, double a1, double a2).
Reimplemented in Fl_PostScript_Graphics_Driver.

void Fl_Graphics_Driver::point (int *x,* int *y*) `[protected]`, `[virtual]`

see fl_point(int x, int y).
Reimplemented in Fl_PostScript_Graphics_Driver.

void Fl_Graphics_Driver::polygon (int *x0,* int *y0,* int *x1,* int *y1,* int *x2,* int *y2*) `[protected]`, `[virtual]`

see fl_polygon(int x0, int y0, int x1, int y1, int x2, int y2).
Reimplemented in Fl_PostScript_Graphics_Driver.

void Fl_Graphics_Driver::polygon (int *x0,* int *y0,* int *x1,* int *y1,* int *x2,* int *y2,* int *x3,* int *y3*) `[protected]`, `[virtual]`

see fl_polygon(int x0, int y0, int x1, int y1, int x2, int y2, int x3, int y3).
Reimplemented in Fl_PostScript_Graphics_Driver.

void Fl_Graphics_Driver::pop_clip ( ) `[protected]`, `[virtual]`

see fl_pop_clip().
Reimplemented in Fl_PostScript_Graphics_Driver.

void Fl_Graphics_Driver::pop_matrix ( ) `[protected]`

see fl_pop_matrix().

void Fl_Graphics_Driver::push_clip (int *x,* int *y,* int *w,* int *h*) `[protected]`, `[virtual]`

see fl_push_clip(int x, int y, int w, int h).
Reimplemented in Fl_PostScript_Graphics_Driver.

void Fl_Graphics_Driver::push_matrix ( ) `[protected]`

see fl_push_matrix().

void Fl_Graphics_Driver::push_no_clip ( ) `[protected]`, `[virtual]`

see fl_push_no_clip().
Reimplemented in Fl_PostScript_Graphics_Driver.

void Fl_Graphics_Driver::rect (int *x,* int *y,* int *w,* int *h*) `[protected]`, `[virtual]`

see fl_rect(int x, int y, int w, int h).
Reimplemented in Fl_PostScript_Graphics_Driver.

void Fl_Graphics_Driver::rectf (int *x,* int *y,* int *w,* int *h*) `[protected]`, `[virtual]`

see fl_rectf(int x, int y, int w, int h).
Reimplemented in Fl_PostScript_Graphics_Driver.

void Fl_Graphics_Driver::restore_clip ( ) `[protected]`

see fl_restore_clip().

void Fl_Graphics_Driver::rotate (double *d*) `[protected]`

see fl_rotate(double d).

virtual void Fl_Graphics_Driver::rtl_draw (const char ∗ *str,* int *n,* int *x,* int *y*) `[inline]`, `[protected]`, `[virtual]`

see fl_rtl_draw(const char ∗str, int n, int x, int y).

Reimplemented in Fl_Xlib_Graphics_Driver, Fl_GDI_Graphics_Driver, Fl_Quartz_Graphics_Driver, and Fl_PostScript_Graphics_Driver.

void Fl_Graphics_Driver::scale (double *x,* double *y*) `[inline]`, `[protected]`

see fl_scale(double x, double y).

void Fl_Graphics_Driver::scale (double *x*) `[inline]`, `[protected]`

see fl_scale(double x).

Fl_Fontsize Fl_Graphics_Driver::size ( ) `[inline]`

see fl_size().

void Fl_Graphics_Driver::text_extents (const char ∗ *t,* int *n,* int & *dx,* int & *dy,* int & *w,* int & *h*) `[virtual]`

see fl_text_extents(const char∗, int n, int& dx, int& dy, int& w, int& h).

Reimplemented in Fl_Xlib_Graphics_Driver, Fl_GDI_Graphics_Driver, Fl_Quartz_Graphics_Driver, and Fl_PostScript_Graphics_Driver.

double Fl_Graphics_Driver::transform_dx (double *x,* double *y*) `[protected]`

see fl_transform_dx(double x, double y).

double Fl_Graphics_Driver::transform_dy (double *x,* double *y*) `[protected]`

see fl_transform_dy(double x, double y).

double Fl_Graphics_Driver::transform_x (double *x,* double *y*) `[protected]`

see fl_transform_x(double x, double y).

double Fl_Graphics_Driver::transform_y (double *x,* double *y*) `[protected]`

see fl_transform_y(double x, double y).

void Fl_Graphics_Driver::transformed_vertex (double *xf,* double *yf*) `[protected]`, `[virtual]`

see fl_transformed_vertex(double xf, double yf).

Reimplemented in Fl_PostScript_Graphics_Driver.

void Fl_Graphics_Driver::translate (double *x*, double *y*) `[inline], [protected]`

see fl_translate(double x, double y).

void Fl_Graphics_Driver::vertex (double *x*, double *y*) `[protected], [virtual]`

see fl_vertex(double x, double y).
Reimplemented in Fl_PostScript_Graphics_Driver.

virtual double Fl_Graphics_Driver::width (const char ∗ *str*, int *n*) `[inline], [virtual]`

see fl_width(const char ∗str, int n).
Reimplemented in Fl_Xlib_Graphics_Driver, Fl_GDI_Graphics_Driver, Fl_Quartz_Graphics_Driver, and Fl_PostScript_Graphics_Driver.

virtual double Fl_Graphics_Driver::width (unsigned int *c*) `[inline], [virtual]`

see fl_width(unsigned int n).
Reimplemented in Fl_Xlib_Graphics_Driver, Fl_GDI_Graphics_Driver, Fl_Quartz_Graphics_Driver, and Fl_PostScript_Graphics_Driver.

void Fl_Graphics_Driver::xyline (int *x*, int *y*, int *x1*) `[protected], [virtual]`

see fl_xyline(int x, int y, int x1).
Reimplemented in Fl_PostScript_Graphics_Driver.

void Fl_Graphics_Driver::xyline (int *x*, int *y*, int *x1*, int *y2*) `[protected], [virtual]`

see fl_xyline(int x, int y, int x1, int y2).
Reimplemented in Fl_PostScript_Graphics_Driver.

void Fl_Graphics_Driver::xyline (int *x*, int *y*, int *x1*, int *y2*, int *x3*) `[protected], [virtual]`

see fl_xyline(int x, int y, int x1, int y2, int x3).
Reimplemented in Fl_PostScript_Graphics_Driver.

void Fl_Graphics_Driver::yxline (int *x*, int *y*, int *y1*) `[protected], [virtual]`

see fl_yxline(int x, int y, int y1).
Reimplemented in Fl_PostScript_Graphics_Driver.

void Fl_Graphics_Driver::yxline (int *x*, int *y*, int *y1*, int *x2*) `[protected], [virtual]`

see fl_yxline(int x, int y, int y1, int x2).
Reimplemented in Fl_PostScript_Graphics_Driver.

void Fl_Graphics_Driver::yxline (int *x*, int *y*, int *y1*, int *x2*, int *y3*) `[protected], [virtual]`

see fl_yxline(int x, int y, int y1, int x2, int y3).
Reimplemented in Fl_PostScript_Graphics_Driver.

31.53.4 Friends And Related Function Documentation

void fl_arc (double *x*, double *y*, double *r*, double *start*, double *end*) `[friend]`

Adds a series of points to the current path on the arc of a circle.
You can get elliptical paths by using scale and rotate before calling fl_arc().

Parameters

in	*x,y,r*	center and radius of circular arc
in	*start,end*	angles of start and end of arc measured in degrees counter-clockwise from 3 o'clock. If `end` is less than `start` then it draws the arc in a clockwise direction.

Examples:

```
// Draw an arc of points
fl_begin_points();
fl_arc(100.0, 100.0, 50.0, 0.0, 180.0);
fl_end_points();

// Draw arc with a line
fl_begin_line();
fl_arc(200.0, 100.0, 50.0, 0.0, 180.0);
fl_end_line();

// Draw filled arc
fl_begin_polygon();
fl_arc(300.0, 100.0, 50.0, 0.0, 180.0);
fl_end_polygon();
```

void fl_arc (int *x*, int *y*, int *w*, int *h*, double *a1*, double *a2*) `[friend]`

Draw ellipse sections using integer coordinates.

These functions match the rather limited circle drawing code provided by X and WIN32. The advantage over using fl_arc with floating point coordinates is that they are faster because they often use the hardware, and they draw much nicer small circles, since the small sizes are often hard-coded bitmaps.

If a complete circle is drawn it will fit inside the passed bounding box. The two angles are measured in degrees counter-clockwise from 3 o'clock and are the starting and ending angle of the arc, `a2` must be greater or equal to `a1`.

fl_arc() draws a series of lines to approximate the arc. Notice that the integer version of fl_arc() has a different number of arguments than the double version fl_arc(double x, double y, double r, double start, double end)

Parameters

in	*x,y,w,h*	bounding box of complete circle
in	*a1,a2*	start and end angles of arc measured in degrees counter-clockwise from 3 o'clock. `a2` must be greater than or equal to `a1`.

void fl_begin_complex_polygon ( ) `[friend]`

Starts drawing a complex filled polygon.

The polygon may be concave, may have holes in it, or may be several disconnected pieces. Call fl_gap() to separate loops of the path.

To outline the polygon, use fl_begin_loop() and replace each fl_gap() with fl_end_loop();fl_begin_loop() pairs.

Note

> For portability, you should only draw polygons that appear the same whether ”even/odd” or ”non-zero” winding rules are used to fill them. Holes should be drawn in the opposite direction to the outside loop.

void fl_begin_points ( ) `[friend]`

Starts drawing a list of points.

Points are added to the list with fl_vertex()

void fl_circle (double *x*, double *y*, double *r*) `[friend]`

fl_circle() is equivalent to fl_arc(x,y,r,0,360), but may be faster.

It must be the *only* thing in the path: if you want a circle as part of a complex polygon you must use fl_arc()

Parameters

`in`	*x,y,r*	center and radius of circle

int fl_clip_box (int *x*, int *y*, int *w*, int *h*, int & *X*, int & *Y*, int & *W*, int & *H*) `[friend]`

Intersects the rectangle with the current clip region and returns the bounding box of the result.

Returns non-zero if the resulting rectangle is different to the original. This can be used to limit the necessary drawing to a rectangle. `W` and `H` are set to zero if the rectangle is completely outside the region.

Parameters

`in`	*x,y,w,h*	position and size of rectangle
`out`	*X,Y,W,H*	position and size of resulting bounding box.

Returns

Non-zero if the resulting rectangle is different to the original.

void fl_clip_region (Fl_Region *r*) `[friend]`

Replaces the top of the clipping stack with a clipping region of any shape.

Fl_Region is an operating system specific type.

Parameters

`in`	*r*	clipping region

void fl_color (Fl_Color *c*) `[friend]`

Sets the color for all subsequent drawing operations.

For colormapped displays, a color cell will be allocated out of `fl_colormap` the first time you use a color. If the colormap fills up then a least-squares algorithm is used to find the closest color. If no valid graphical context (fl_gc) is available, the foreground is not set for the current window.

Parameters

`in`	*c*	color

void fl_color (uchar *r*, uchar *g*, uchar *b*) `[friend]`

Sets the color for all subsequent drawing operations.

The closest possible match to the RGB color is used. The RGB color is used directly on TrueColor displays. For colormap visuals the nearest index in the gray ramp or color cube is used. If no valid graphical context (fl_gc) is available, the foreground is not set for the current window.

Parameters

`in`	*r,g,b*	color components

FL_EXPORT void fl_copy_offscreen (int *x*, int *y*, int *w*, int *h*, Fl_Offscreen *pixmap*, int *srcx*, int *srcy*) `[friend]`

Copy a rectangular area of the given offscreen buffer into the current drawing destination.

Parameters

x,y	position where to draw the copied rectangle
w,h	size of the copied rectangle
pixmap	offscreen buffer containing the rectangle to copy
srcx,srcy	origin in offscreen buffer of rectangle to copy

void fl_curve (double *X0*, double *Y0*, double *X1*, double *Y1*, double *X2*, double *Y2*, double *X3*, double *Y3*) `[friend]`

Adds a series of points on a Bezier curve to the path.

The curve ends (and two of the points) are at X0,Y0 and X3,Y3.

Parameters

`in`	*X0,Y0*	curve start point
`in`	*X1,Y1*	curve control point
`in`	*X2,Y2*	curve control point
`in`	*X3,Y3*	curve end point

void fl_draw (int *angle*, const char ∗ *str*, int *n*, int *x*, int *y*) `[friend]`

Draws at the given `x`, `y` location a UTF-8 string of length `n` bytes rotating `angle` degrees counter-clockwise.

Note

When using X11 (Unix, Linux, Cygwin et al.) this needs Xft to work. Under plain X11 (w/o Xft) rotated text is not supported by FLTK. A warning will be issued to stderr at runtime (only once) if you use this method with an angle other than 0.

void fl_draw_image (const uchar ∗ *buf*, int *X*, int *Y*, int *W*, int *H*, int *D* = *3*, int *L* = `0`) `[friend]`

Draws an 8-bit per color RGB or luminance image.

Parameters

`in`	*buf*	points at the "r" data of the top-left pixel. Color data must be in `r,g,b` order. Luminance data is only one `gray` byte.
`in`	*X,Y*	position where to put top-left corner of image
`in`	*W,H*	size of the image
`in`	*D*	delta to add to the pointer between pixels. It may be any value greater than or equal to 1, or it can be negative to flip the image horizontally
`in`	*L*	delta to add to the pointer between lines (if 0 is passed it uses `W ∗ D`), and may be larger than `W ∗ D` to crop data, or negative to flip the image vertically

It is highly recommended that you put the following code before the first `show()` of *any* window in your program to get rid of the dithering if possible:

```
Fl::visual(FL_RGB);
```

Gray scale (1-channel) images may be drawn. This is done if `abs(D)` is less than 3, or by calling fl_draw_image_mono(). Only one 8-bit sample is used for each pixel, and on screens with different numbers of bits for red, green, and blue only gray colors are used. Setting `D` greater than 1 will let you display one channel of a color image.

Note:

The X version does not support all possible visuals. If FLTK cannot draw the image in the current visual it will abort. FLTK supports any visual of 8 bits or less, and all common TrueColor visuals up to 32 bits.

void fl_draw_image (Fl_Draw_Image_Cb *cb*, void ∗ *data*, int *X*, int *Y*, int *W*, int *H*, int *D* = `3`) `[friend]`

Draws an image using a callback function to generate image data.

You can generate the image as it is being drawn, or do arbitrary decompression of stored data, provided it can be decompressed to individual scan lines easily.

Parameters

`in`	*cb*	callback function to generate scan line data
`in`	*data*	user data passed to callback function
`in`	*X,Y*	screen position of top left pixel
`in`	*W,H*	image width and height
`in`	*D*	data size in bytes (must be greater than 0)

See Also

fl_draw_image(const uchar∗ buf, int X,int Y,int W,int H, int D, int L)

The callback function `cb` is called with the `void∗ data` user data pointer to allow access to a structure of information about the image, and the `x`, `y`, and `w` of the scan line desired from the image. 0,0 is the upper-left corner of the image, not `x`, `y`. A pointer to a buffer to put the data into is passed. You must copy `w` pixels from scanline `y`, starting at pixel `x`, to this buffer.

Due to cropping, less than the whole image may be requested. So `x` may be greater than zero, the first `y` may be greater than zero, and `w` may be less than `W`. The buffer is long enough to store the entire `W ∗ D` pixels, this is for convenience with some decompression schemes where you must decompress the entire line at once: decompress it into the buffer, and then if `x` is not zero, copy the data over so the `x'th` pixel is at the start of the buffer.

You can assume the `y's` will be consecutive, except the first one may be greater than zero.

If `D` is 4 or more, you must fill in the unused bytes with zero.

void fl_draw_image_mono (const uchar ∗ *buf*, int *X*, int *Y*, int *W*, int *H*, int *D* = `1`, int *L* = `0`) `[friend]`

Draws a gray-scale (1 channel) image.

See Also

fl_draw_image(const uchar∗ buf, int X,int Y,int W,int H, int D, int L)

FL_EXPORT void fl_draw_image_mono (Fl_Draw_Image_Cb *cb*, void ∗ *data*, int *X*, int *Y*, int *W*, int *H*, int *D* = `1`) `[friend]`

Draws a gray-scale image using a callback function to generate image data.

See Also

fl_draw_image(Fl_Draw_Image_Cb cb, void∗ data, int X,int Y,int W,int H, int D)

void fl_font (Fl_Font *face*, Fl_Fontsize *size*) `[friend]`

Sets the current font, which is then used in various drawing routines.

You may call this outside a draw context if necessary to call fl_width(), but on X this will open the display.

The font is identified by a `face` and a `size`. The size of the font is measured in pixels and not "points". Lines should be spaced `size` pixels apart or more.

void fl_gap ( ) `[friend]`

Call fl_gap() to separate loops of the path.

It is unnecessary but harmless to call fl_gap() before the first vertex, after the last vertex, or several times in a row.

void fl_line_style (int *style*, int *width* = *0*, char ∗ *dashes* = *0*) `[friend]`

Sets how to draw lines (the "pen").

If you change this it is your responsibility to set it back to the default using `fl_line_style(0)`.

Parameters

`in`	*style*	A bitmask which is a bitwise-OR of a line style, a cap style, and a join style. If you don't specify a dash type you will get a solid line. If you don't specify a cap or join type you will get a system-defined default of whatever value is fastest.
`in`	*width*	The thickness of the lines in pixels. Zero results in the system defined default, which on both X and Windows is somewhat different and nicer than 1.
`in`	*dashes*	A pointer to an array of dash lengths, measured in pixels. The first location is how long to draw a solid portion, the next is how long to draw the gap, then the solid, etc. It is terminated with a zero-length entry. A `NULL` pointer or a zero-length array results in a solid line. Odd array sizes are not supported and result in undefined behavior.

Note

Because of how line styles are implemented on Win32 systems, you *must* set the line style *after* setting the drawing color. If you set the color after the line style you will lose the line style settings. The `dashes` array does not work under Windows 95, 98 or Me, since those operating systems do not support complex line styles.

void fl_mult_matrix (double *a*, double *b*, double *c*, double *d*, double *x*, double *y*) `[friend]`

Concatenates another transformation onto the current one.

Parameters

`in`	*a,b,c,d,x,y*	transformation matrix elements such that `X' = aX + cY + x` and `Y' = bX +dY + y`

int fl_not_clipped (int *x*, int *y*, int *w*, int *h*) `[friend]`

Does the rectangle intersect the current clip region?

Parameters

`in`	*x,y,w,h*	position and size of rectangle

Returns

non-zero if any of the rectangle intersects the current clip region. If this returns 0 you don't have to draw the object.

Note

Under X this returns 2 if the rectangle is partially clipped, and 1 if it is entirely inside the clip region.

void fl_pie (int *x*, int *y*, int *w*, int *h*, double *a1*, double *a2*) `[friend]`

Draw filled ellipse sections using integer coordinates.

Like fl_arc(), but fl_pie() draws a filled-in pie slice. This slice may extend outside the line drawn by fl_arc(); to avoid this use w - 1 and h - 1.

Parameters

`in`	*x,y,w,h*	bounding box of complete circle
`in`	*a1,a2*	start and end angles of arc measured in degrees counter-clockwise from 3 o'clock. `a2` must be greater than or equal to `a1`.

void fl_polygon (int *x0*, int *y0*, int *x1*, int *y1*, int *x2*, int *y2*) `[friend]`

Fills a 3-sided polygon.

The polygon must be convex.

void fl_polygon (int *x0*, int *y0*, int *x1*, int *y1*, int *x2*, int *y2*, int *x3*, int *y3*) `[friend]`

Fills a 4-sided polygon.

The polygon must be convex.

void fl_pop_clip ( ) `[friend]`

Restores the previous clip region.

You must call fl_pop_clip() once for every time you call fl_push_clip(). Unpredictable results may occur if the clip stack is not empty when you return to FLTK.

void fl_push_clip (int *x*, int *y*, int *w*, int *h*) `[friend]`

Intersects the current clip region with a rectangle and pushes this new region onto the stack.

Parameters

`in`	*x,y,w,h*	position and size

void fl_push_matrix ( ) `[friend]`

Saves the current transformation matrix on the stack.

The maximum depth of the stack is 32.

void fl_rect (int *x*, int *y*, int *w*, int *h*) `[friend]`

Draws a 1-pixel border *inside* the given bounding box.

This function is meant for quick drawing of simple boxes. The behavior is undefined for line widths that are not 1.

void fl_rotate (double *d*) `[friend]`

Concatenates rotation transformation onto the current one.

Parameters

in	*d*	- rotation angle, counter-clockwise in degrees (not radians)

void fl_scale (double *x*, double *y*) `[friend]`

Concatenates scaling transformation onto the current one.
Parameters

in	*x,y*	scale factors in x-direction and y-direction

void fl_scale (double *x*) `[friend]`

Concatenates scaling transformation onto the current one.
Parameters

in	*x*	scale factor in both x-direction and y-direction

double fl_transform_dx (double *x*, double *y*) `[friend]`

Transforms distance using current transformation matrix.
Parameters

in	*x,y*	coordinate

double fl_transform_dy (double *x*, double *y*) `[friend]`

Transforms distance using current transformation matrix.
Parameters

in	*x,y*	coordinate

double fl_transform_x (double *x*, double *y*) `[friend]`

Transforms coordinate using the current transformation matrix.
Parameters

in	*x,y*	coordinate

double fl_transform_y (double *x*, double *y*) `[friend]`

Transforms coordinate using the current transformation matrix.
Parameters

in	*x,y*	coordinate

void fl_transformed_vertex (double *xf*, double *yf*) `[friend]`

Adds coordinate pair to the vertex list without further transformations.

Parameters

in	*xf,yf*	transformed coordinate

void fl_translate (double *x*, double *y*) [friend]

Concatenates translation transformation onto the current one.

Parameters

in	*x,y*	translation factor in x-direction and y-direction

void fl_vertex (double *x*, double *y*) [friend]

Adds a single vertex to the current path.

Parameters

in	*x,y*	coordinate

The documentation for this class was generated from the following files:

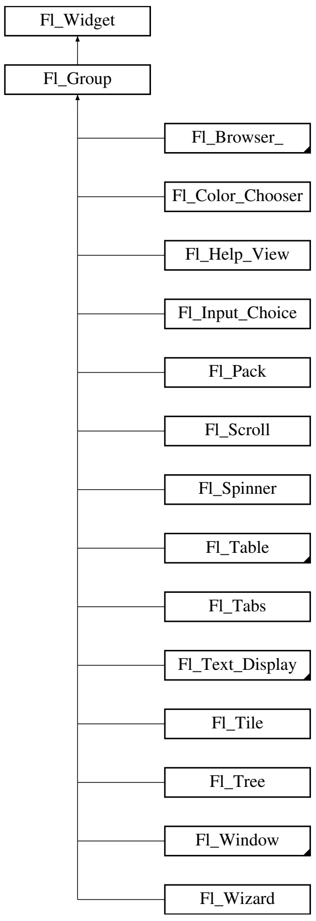
- Fl_Device.H
- fl_arc.cxx
- fl_arci.cxx
- fl_curve.cxx
- Fl_Device.cxx
- Fl_Double_Window.cxx
- Fl_Image.cxx
- fl_line_style.cxx
- fl_rect.cxx
- fl_vertex.cxx

31.54 Fl_Group Class Reference

The Fl_Group class is the FLTK container widget.

```
#include <Fl_Group.H>
```

Inheritance diagram for Fl_Group:



Public Member Functions

- Fl_Widget ∗& _ddfdesign_kludge ()

  This is for forms compatibility only.
- void add (Fl_Widget &)

  The widget is removed from its current group (if any) and then added to the end of this group.
- void add (Fl_Widget ∗o)

  See void Fl_Group::add(Fl_Widget &w)
- void add_resizable (Fl_Widget &o)

  Adds a widget to the group and makes it the resizable widget.
- Fl_Widget ∗const ∗ array () const

  Returns a pointer to the array of children.
- virtual Fl_Group ∗ as_group ()

  Returns an Fl_Group pointer if this widget is an Fl_Group.
- void begin ()

  Sets the current group so you can build the widget tree by just constructing the widgets.
- Fl_Widget ∗ child (int n) const

  Returns array()[n].
- int children () const

  Returns how many child widgets the group has.

- void clear ()

  Deletes all child widgets from memory recursively.
- void clip_children (int c)

  Controls whether the group widget clips the drawing of child widgets to its bounding box.
- unsigned int clip_children ()

  Returns the current clipping mode.
- void end ()

  Exactly the same as current(this->parent()).
- int find (const Fl_Widget ∗) const

  Searches the child array for the widget and returns the index.
- int find (const Fl_Widget &o) const

  See int Fl_Group::find(const Fl_Widget ∗w) const.
- Fl_Group (int, int, int, int, const char ∗=0)

  Creates a new Fl_Group widget using the given position, size, and label string.
- void focus (Fl_Widget ∗W)
- void forms_end ()

  This is for forms compatibility only.
- int handle (int)

  Handles the specified event.
- void init_sizes ()

  Resets the internal array of widget sizes and positions.
- void insert (Fl_Widget &, int i)

  The widget is removed from its current group (if any) and then inserted into this group.
- void insert (Fl_Widget &o, Fl_Widget ∗before)

  This does insert(w, find(before)).
- void remove (int index)

  Removes the widget at `index` from the group but does not delete it.
- void remove (Fl_Widget &)

  Removes a widget from the group but does not delete it.
- void remove (Fl_Widget ∗o)

  Removes the widget `o` from the group.
- void resizable (Fl_Widget &o)

  See void Fl_Group::resizable(Fl_Widget ∗box)
- void resizable (Fl_Widget ∗o)

  The resizable widget defines the resizing box for the group.
- Fl_Widget ∗ resizable () const

  See void Fl_Group::resizable(Fl_Widget ∗box)
- void resize (int, int, int, int)

  Resizes the Fl_Group widget and all of its children.
- virtual ∼Fl_Group ()

  The destructor also deletes all the children.

Static Public Member Functions

- static Fl_Group ∗ current ()

  Returns the currently active group.
- static void current (Fl_Group ∗g)

  Sets the current group.

Protected Member Functions

- void draw ()

  Draws the widget.
- void draw_child (Fl_Widget &widget) const

  Forces a child to redraw.
- void draw_children ()

  Draws all children of the group.
- void draw_outside_label (const Fl_Widget &widget) const

  Parents normally call this to draw outside labels of child widgets.
- int ∗ sizes ()

  Returns the internal array of widget sizes and positions.
- void update_child (Fl_Widget &widget) const

  Draws a child only if it needs it.

Additional Inherited Members

31.54.1 Detailed Description

The Fl_Group class is the FLTK container widget.

It maintains an array of child widgets. These children can themselves be any widget including Fl_Group. The most important subclass of Fl_Group is Fl_Window, however groups can also be used to control radio buttons or to enforce resize behavior.

The tab and arrow keys are used to move the focus between widgets of this group, and to other groups. The only modifier grabbed is shift (for shift-tab), so that ctrl-tab, alt-up, and such are free for the app to use as shortcuts.

31.54.2 Constructor & Destructor Documentation

Fl_Group::Fl_Group (int *X*, int *Y*, int *W*, int *H*, const char ∗ *l* = 0)

Creates a new Fl_Group widget using the given position, size, and label string.

The default boxtype is FL_NO_BOX.

Fl_Group::∼Fl_Group ( ) `[virtual]`

The destructor *also deletes all the children*.

This allows a whole tree to be deleted at once, without having to keep a pointer to all the children in the user code.

It is allowed that the Fl_Group and all of its children are automatic (local) variables, but you must declare the Fl_Group *first*, so that it is destroyed last.

If you add static or automatic (local) variables to an Fl_Group, then it is your responsibility to remove (or delete) all such static or automatic child widgets ***before** destroying* the group - otherwise the child widgets' destructors would be called twice!

31.54.3 Member Function Documentation

Fl_Widget ∗const ∗ Fl_Group::array ( ) const

Returns a pointer to the array of children.

This pointer is only valid until the next time a child is added or removed.

virtual Fl_Group∗ Fl_Group::as_group ( ) `[inline]`, `[virtual]`

Returns an Fl_Group pointer if this widget is an Fl_Group.

Use this method if you have a widget (pointer) and need to know whether this widget is derived from Fl_Group. If it returns non-NULL, then the widget in question is derived from Fl_Group, and you can use the returned pointer to access its children or other Fl_Group-specific methods.

Example:

```
void my_callback (Fl_Widget *w, void *) {
  Fl_Group *g = w->as_group();
  if (g)
    printf ("This group has %d children\n",g->children());
  else
    printf ("This widget is not a group!\n");
}
```

Return values

NULL	if this widget is not derived from Fl_Group.

Note

This method is provided to avoid dynamic_cast.

See Also

Fl_Widget::as_window(), Fl_Widget::as_gl_window()

Reimplemented from Fl_Widget.

void Fl_Group::begin ( )

Sets the current group so you can build the widget tree by just constructing the widgets.

begin() is automatically called by the constructor for Fl_Group (and thus for Fl_Window as well). begin() *is exactly the same as* current(this). *Don't forget to end() the group or window!*

Fl_Widget∗ Fl_Group::child (int *n*) const `[inline]`

Returns array()[n].

No range checking is done!

void Fl_Group::clear ( )

Deletes all child widgets from memory recursively.

This method differs from the remove() method in that it affects all child widgets and deletes them from memory.

void Fl_Group::clip_children (int *c*) `[inline]`

Controls whether the group widget clips the drawing of child widgets to its bounding box.

Set `c` to 1 if you want to clip the child widgets to the bounding box.

The default is to not clip (0) the drawing of child widgets.

unsigned int Fl_Group::clip_children ( ) `[inline]`

Returns the current clipping mode.

Returns

true, if clipping is enabled, false otherwise.

See Also

void Fl_Group::clip_children(int c)

Fl_Group ∗ Fl_Group::current ( ) [static]

Returns the currently active group.

The Fl_Widget constructor automatically does current()->add(widget) if this is not null. To prevent new widgets from being added to a group, call Fl_Group::current(0).

void Fl_Group::current (Fl_Group ∗ *g*) [static]

Sets the current group.

See Also

Fl_Group::current()

void Fl_Group::draw ( ) [protected], [virtual]

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call redraw() instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw() method*, e.g. for an embedded scrollbar, you can do it (because draw() is virtual) like this:

```
Fl_Widget *s = &scroll;              // scroll is an embedded Fl_Scrollbar
s->draw();                    // calls Fl_Scrollbar::draw()
```

Implements Fl_Widget.

Reimplemented in Fl_Table, Fl_Tree, Fl_Text_Display, Fl_Help_View, Fl_Tabs, Fl_Window, Fl_Scroll, Fl_Pack, and Fl_Glut_Window.

void Fl_Group::draw_child (Fl_Widget & *widget*) const [protected]

Forces a child to redraw.

This draws a child widget, if it is not clipped. The damage bits are cleared after drawing.

void Fl_Group::draw_children ( ) [protected]

Draws all children of the group.

This is useful, if you derived a widget from Fl_Group and want to draw a special border or background. You can call draw_children() from the derived draw() method after drawing the box, border, or background.

void Fl_Group::draw_outside_label (const Fl_Widget & *widget*) const [protected]

Parents normally call this to draw outside labels of child widgets.

void Fl_Group::end ( )

Exactly the same as current(this->parent()).

Any new widgets added to the widget tree will be added to the parent of the group.

int Fl_Group::find (const Fl_Widget ∗ *o*) const

Searches the child array for the widget and returns the index.

Returns children() if the widget is NULL or not found.

void Fl_Group::focus (Fl_Widget ∗ *W*) `[inline]`

Deprecated This is for backwards compatibility only. You should use *W->take_focus*() instead.

See Also

Fl_Widget::take_focus();

int Fl_Group::handle (int *event*) `[virtual]`

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited handle() method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters

`in`	*event*	the kind of event received

Return values

0	if the event was not used or understood
1	if the event was used and can be deleted

See Also

Fl_Event

Reimplemented from Fl_Widget.

Reimplemented in Fl_Tree, Fl_Table, Fl_Help_View, Fl_Window, Fl_Tabs, Fl_Text_Display, Fl_Scroll, Fl_Spinner, Fl_Table_Row, Fl_Text_Editor, Fl_Glut_Window, and Fl_Tile.

void Fl_Group::init_sizes ( )

Resets the internal array of widget sizes and positions.

The Fl_Group widget keeps track of the original widget sizes and positions when resizing occurs so that if you resize a window back to its original size the widgets will be in the correct places. If you rearrange the widgets in your group, call this method to register the new arrangement with the Fl_Group that contains them.

If you add or remove widgets, this will be done automatically.

Note

The internal array of widget sizes and positions will be allocated and filled when the next resize() occurs.

See Also

sizes()

void Fl_Group::insert (Fl_Widget & *o*, int *index*)

The widget is removed from its current group (if any) and then inserted into this group.

It is put at index n - or at the end, if n >= children(). This can also be used to rearrange the widgets inside a group.

void Fl_Group::insert (Fl_Widget & *o*, Fl_Widget ∗ *before*) `[inline]`

This does insert(w, find(before)).

This will append the widget if `before` is not in the group.

void Fl_Group::remove (int *index*)

Removes the widget at `index` from the group but does not delete it.

This method does nothing if `index` is out of bounds.

This method differs from the clear() method in that it only affects a single widget and does not delete it from memory.

Since

FLTK 1.3.0

void Fl_Group::remove (Fl_Widget & *o*)

Removes a widget from the group but does not delete it.

This method does nothing if the widget is not a child of the group.

This method differs from the clear() method in that it only affects a single widget and does not delete it from memory.

Note

If you have the child's index anyway, use remove(int index) instead, because this doesn't need a child lookup in the group's table of children. This can be much faster, if there are lots of children.

void Fl_Group::remove (Fl_Widget ∗ *o*) `[inline]`

Removes the widget `o` from the group.

See Also

void remove(Fl_Widget&)

void Fl_Group::resizable (Fl_Widget ∗ *o*) `[inline]`

The resizable widget defines the resizing box for the group.

When the group is resized it calculates a new size and position for all of its children. Widgets that are horizontally or vertically inside the dimensions of the box are scaled to the new size. Widgets outside the box are moved.

In these examples the gray area is the resizable:

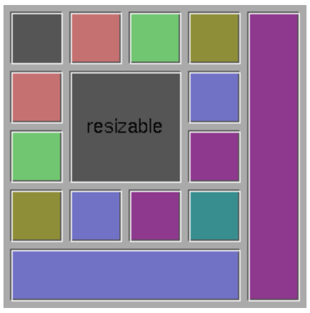


Figure 31.14: before resize

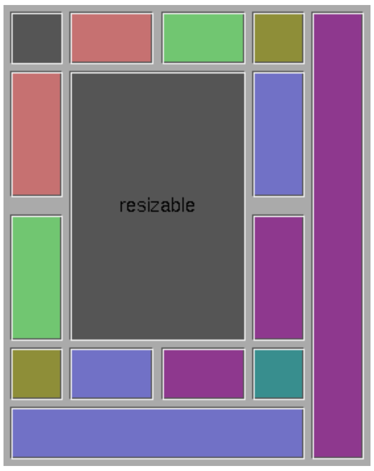


Figure 31.15: after resize

The resizable may be set to the group itself, in which case all the contents are resized. This is the default value for Fl_Group, although NULL is the default for Fl_Window and Fl_Pack.

If the resizable is NULL then all widgets remain a fixed size and distance from the top-left corner.

It is possible to achieve any type of resize behavior by using an invisible Fl_Box as the resizable and/or by using a hierarchy of child Fl_Group's.

void Fl_Group::resize (int *X*, int *Y*, int *W*, int *H*) `[virtual]`

Resizes the Fl_Group widget and all of its children.

The Fl_Group widget first resizes itself, and then it moves and resizes all its children according to the rules documented for Fl_Group::resizable(Fl_Widget∗)

See Also

- Fl_Group::resizable(Fl_Widget∗)
- Fl_Group::resizable()
- Fl_Widget::resize(int,int,int,int)

Reimplemented from Fl_Widget.

Reimplemented in Fl_Table, Fl_Tree, Fl_Help_View, Fl_Text_Display, Fl_Window, Fl_Input_Choice, Fl_Spinner, Fl_Scroll, Fl_Overlay_Window, and Fl_Tile.

int ∗ Fl_Group::sizes ( ) `[protected]`

Returns the internal array of widget sizes and positions.

If the sizes() array does not exist, it will be allocated and filled with the current widget sizes and positions.

Note

> You should never need to use this method directly, unless you have special needs to rearrange the children of a Fl_Group. Fl_Tile uses this to rearrange its widget positions.

See Also

> init_sizes()

Todo Should the internal representation of the sizes() array be documented?

void Fl_Group::update_child (Fl_Widget & *widget*) const `[protected]`

Draws a child only if it needs it.

This draws a child widget, if it is not clipped *and* if any damage() bits are set. The damage bits are cleared after drawing.

See Also

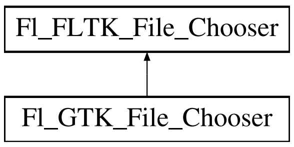
> Fl_Group::draw_child(Fl_Widget& widget) const

The documentation for this class was generated from the following files:

- Fl_Group.H
- Fl_Group.cxx
- forms_compatability.cxx

31.55 Fl_GTK_File_Chooser Class Reference

Inheritance diagram for Fl_GTK_File_Chooser:



Friends

- class **Fl_Native_File_Chooser**

Additional Inherited Members

The documentation for this class was generated from the following files:

- Fl_Native_File_Chooser.H
- Fl_Native_File_Chooser_GTK.cxx

31.56 Fl_Help_Block Struct Reference

Public Attributes

- Fl_Color **bgcolor**
- uchar **border**
- const char ∗ **end**
- int **h**
- int **line** [32]
- const char ∗ **start**
- int **w**
- int **x**
- int **y**

The documentation for this struct was generated from the following file:

- Fl_Help_View.H

31.57 Fl_Help_Dialog Class Reference

The Fl_Help_Dialog widget displays a standard help dialog window using the Fl_Help_View widget.

Public Member Functions

- Fl_Help_Dialog ()

  The constructor creates the dialog pictured above.
- int h ()

  Returns the position and size of the help dialog.
- void hide ()

  Hides the Fl_Help_Dialog window.
- void load (const char ∗f)

  Loads the specified HTML file into the Fl_Help_View widget.
- void position (int xx, int yy)

  Set the screen position of the dialog.
- void resize (int xx, int yy, int ww, int hh)

  Change the position and size of the dialog.
- void show ()

  Shows the Fl_Help_Dialog window.
- void show (int argc, char ∗∗argv)

  Shows the main Help Dialog Window Delegates call to encapsulated window_ void Fl_Window::show(int argc, char ∗∗argv) instance method.
- void textsize (Fl_Fontsize s)

  Sets or gets the default text size for the help view.
- Fl_Fontsize textsize ()

  Sets or gets the default text size for the help view.
- void topline (const char ∗n)

  Sets the top line in the Fl_Help_View widget to the named or numbered line.
- void topline (int n)

  Sets the top line in the Fl_Help_View widget to the named or numbered line.
- void value (const char ∗f)

The first form sets the current buffer to the string provided and reformats the text.

- const char ∗ value () const

  The first form sets the current buffer to the string provided and reformats the text.

- int visible ()

  Returns 1 if the Fl_Help_Dialog window is visible.

- int w ()

  Returns the position and size of the help dialog.

- int x ()

  Returns the position and size of the help dialog.

- int y ()

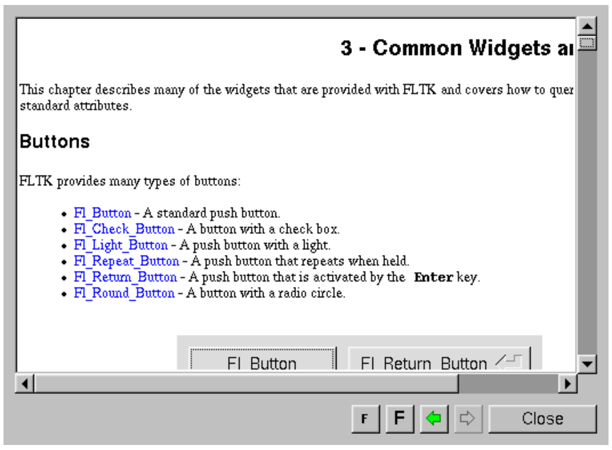
  Returns the position and size of the help dialog.

- ∼Fl_Help_Dialog ()

  The destructor destroys the widget and frees all memory that has been allocated for the current file.

31.57.1 Detailed Description

The Fl_Help_Dialog widget displays a standard help dialog window using the Fl_Help_View widget.



3 - Common Widgets a

This chapter describes many of the widgets that are provided with FLTK and covers how to quer
standard attributes.

Buttons

FLTK provides many types of buttons:

- Fl_Button - A standard push button.
- Fl_Check_Button - A button with a check box.
- Fl_Light_Button - A push button with a light.
- Fl_Repeat_Button - A push button that repeats when held.
- Fl_Return_Button - A push button that is activated by the **Enter** key.
- Fl_Round_Button - A button with a radio circle.

Figure 31.16: Fl_Help_Dialog

31.57.2 Constructor & Destructor Documentation

Fl_Help_Dialog::Fl_Help_Dialog ( )

The constructor creates the dialog pictured above.

31.57.3 Member Function Documentation

int Fl_Help_Dialog::h ( )

Returns the position and size of the help dialog.

void Fl_Help_Dialog::hide ( )

Hides the Fl_Help_Dialog window.

void Fl_Help_Dialog::load (const char ∗ *f*)

Loads the specified HTML file into the Fl_Help_View widget.

The filename can also contain a target name ("filename.html#target").

void Fl_Help_Dialog::position (int *x*, int *y*)

Set the screen position of the dialog.

void Fl_Help_Dialog::resize (int *xx*, int *yy*, int *ww*, int *hh*)

Change the position and size of the dialog.

void Fl_Help_Dialog::show ( )

Shows the Fl_Help_Dialog window.

Shows the main Help Dialog Window Delegates call to encapsulated window_ void Fl_Window::show() method.

void Fl_Help_Dialog::textsize (Fl_Fontsize *s*)

Sets or gets the default text size for the help view.

Sets the internal Fl_Help_View instance text size.

Delegates call to encapsulated view_ void Fl_Help_View::textsize(Fl_Fontsize s) instance method

uchar Fl_Help_Dialog::textsize ( )

Sets or gets the default text size for the help view.

void Fl_Help_Dialog::value (const char ∗ *v*)

The first form sets the current buffer to the string provided and reformats the text.

It also clears the history of the "back" and "forward" buttons. The second form returns the current buffer contents.

const char ∗ Fl_Help_Dialog::value ( ) const

The first form sets the current buffer to the string provided and reformats the text.

It also clears the history of the "back" and "forward" buttons. The second form returns the current buffer contents.

int Fl_Help_Dialog::visible ( )

Returns 1 if the Fl_Help_Dialog window is visible.

int Fl_Help_Dialog::w ( )

Returns the position and size of the help dialog.

int Fl_Help_Dialog::x ( )

Returns the position and size of the help dialog.

int Fl_Help_Dialog::y ( )

Returns the position and size of the help dialog.

The documentation for this class was generated from the following files:

- Fl_Help_Dialog.H
- Fl_Help_Dialog.cxx
- Fl_Help_Dialog_Dox.cxx

31.58 Fl_Help_Font_Stack Struct Reference

Public Member Functions

- size_t count () const

  Gets the current count of font style elements in the stack.
- Fl_Help_Font_Stack ()

  font stack construction, initialize attributes.
- void **init** (Fl_Font f, Fl_Fontsize s, Fl_Color c)
- void pop (Fl_Font &f, Fl_Fontsize &s, Fl_Color &c)

  Pops from the stack the font style triplet and calls fl_font() & fl_color() adequately.
- void push (Fl_Font f, Fl_Fontsize s, Fl_Color c)

  Pushes the font style triplet on the stack, also calls fl_font() & fl_color() adequately.
- void top (Fl_Font &f, Fl_Fontsize &s, Fl_Color &c)

  Gets the top (current) element on the stack.

Protected Attributes

- Fl_Help_Font_Style elts_ [100]

  font elements
- size_t nfonts_

  current number of fonts in stack

31.58.1 Constructor & Destructor Documentation

Fl_Help_Font_Stack::Fl_Help_Font_Stack ( ) `[inline]`

font stack construction, initialize attributes.

31.58.2 Member Function Documentation

size_t Fl_Help_Font_Stack::count ( ) const `[inline]`

Gets the current count of font style elements in the stack.

void Fl_Help_Font_Stack::top (Fl_Font & *f*, Fl_Fontsize & *s*, Fl_Color & *c*) `[inline]`

Gets the top (current) element on the stack.

The documentation for this struct was generated from the following file:

- Fl_Help_View.H

31.59 Fl_Help_Font_Style Struct Reference

Fl_Help_View font stack element definition.

```
#include <Fl_Help_View.H>
```

Public Member Functions

- **Fl_Help_Font_Style** (Fl_Font afont, Fl_Fontsize asize, Fl_Color acolor)
- void get (Fl_Font &afont, Fl_Fontsize &asize, Fl_Color &acolor)

  Gets current font attributes.
- void set (Fl_Font afont, Fl_Fontsize asize, Fl_Color acolor)

  Sets current font attributes.

Public Attributes

- Fl_Color c

  Font Color.
- Fl_Font f

  Font.
- Fl_Fontsize s

  Font Size.

31.59.1 Detailed Description

Fl_Help_View font stack element definition.

The documentation for this struct was generated from the following file:

- Fl_Help_View.H

31.60 Fl_Help_Link Struct Reference

Definition of a link for the html viewer.

```
#include <Fl_Help_View.H>
```

Public Attributes

- char filename [192]

  Reference filename.
- int h

  Height of link text.
- char name [32]

  Link target (blank if none)
- int w

  Width of link text.
- int x

  X offset of link text.
- int y

  Y offset of link text.

31.60.1 Detailed Description

Definition of a link for the html viewer.

The documentation for this struct was generated from the following file:

- Fl_Help_View.H

31.61 Fl_Help_Target Struct Reference

Fl_Help_Target structure.

`#include <Fl_Help_View.H>`

Public Attributes

- char name [32]

  Target name.
- int y

  Y offset of target.

31.61.1 Detailed Description

Fl_Help_Target structure.

The documentation for this struct was generated from the following file:

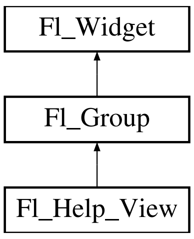
- Fl_Help_View.H

31.62 Fl_Help_View Class Reference

The Fl_Help_View widget displays HTML text.

`#include <Fl_Help_View.H>`

Inheritance diagram for Fl_Help_View:



Public Member Functions

- void clear_selection ()

  Removes the current text selection.
- const char ∗ directory () const

  Returns the current directory for the text in the buffer.
- const char ∗ filename () const

  Returns the current filename for the text in the buffer.
- int find (const char ∗s, int p=0)

  Finds the specified string `s` *at starting position* `p`.
- Fl_Help_View (int xx, int yy, int ww, int hh, const char ∗l=0)

The constructor creates the Fl_Help_View widget at the specified position and size.

- int handle (int)

  Handles events in the widget.

- void leftline (int)

  Scrolls the text to the indicated position, given a pixel column.

- int leftline () const

  Gets the left position in pixels.

- void link (Fl_Help_Func ∗fn)

  This method assigns a callback function to use when a link is followed or a file is loaded (via Fl_Help_View-::load()) that requires a different file or path.

- int load (const char ∗f)

  Loads the specified file.

- void resize (int, int, int, int)

  Resizes the help widget.

- int scrollbar_size () const

  Gets the current size of the scrollbars' troughs, in pixels.

- void scrollbar_size (int newSize)

  Sets the pixel size of the scrollbars' troughs to `newSize`*, in pixels.*

- void select_all ()

  Selects all the text in the view.

- int size () const

  Gets the size of the help view.

- void **size** (int W, int H)
- void textcolor (Fl_Color c)

  Sets the default text color.

- Fl_Color textcolor () const

  Returns the current default text color.

- void textfont (Fl_Font f)

  Sets the default text font.

- Fl_Font textfont () const

  Returns the current default text font.

- void textsize (Fl_Fontsize s)

  Sets the default text size.

- Fl_Fontsize textsize () const

  Gets the default text size.

- const char ∗ title ()

  Returns the current document title, or NULL if there is no title.

- void topline (const char ∗n)

  Scrolls the text to the indicated position, given a named destination.

- void topline (int)

  Scrolls the text to the indicated position, given a pixel line.

- int topline () const

  Returns the current top line in pixels.

- void value (const char ∗val)

  Sets the current help text buffer to the string provided and reformats the text.

- const char ∗ value () const

  Returns the current buffer contents.

- ∼Fl_Help_View ()

  Destroys the Fl_Help_View widget.

Protected Member Functions

- void draw ()

  Draws the Fl_Help_View widget.

Additional Inherited Members

31.62.1 Detailed Description

The Fl_Help_View widget displays HTML text.

Most HTML 2.0 elements are supported, as well as a primitive implementation of tables. GIF, JPEG, and PNG images are displayed inline.

Supported HTML tags:

- A: HREF/NAME
- B
- BODY: BGCOLOR/TEXT/LINK
- BR
- CENTER
- CODE
- DD
- DL
- DT
- EM
- FONT: COLOR/SIZE/FACE=(helvetica/arial/sans/times/serif/symbol/courier)
- H1/H2/H3/H4/H5/H6
- HEAD
- HR
- I
- IMG: SRC/WIDTH/HEIGHT/ALT
- KBD
- LI
- OL
- P
- PRE
- STRONG
- TABLE: TH/TD/TR/BORDER/BGCOLOR/COLSPAN/ALIGN=CENTER|RIGHT|LEFT
- TITLE
- TT

- U
- UL
- VAR

Supported color names:

- black,red,green,yellow,blue,magenta,fuchsia,cyan,aqua,white,gray,grey,lime,maroon,navy,olive,purple,silver,teal.

Supported urls:

- Internal: file:
- External: http: ftp: https: ipp: mailto: news:

Quoted char names:

- Aacute aacute Acirc acirc acute AElig aelig Agrave agrave amp Aring aring Atilde atilde Auml auml
- brvbar bull
- Ccedil ccedil cedil cent copy curren
- deg divide
- Eacute eacute Ecirc ecirc Egrave egrave ETH eth Euml euml euro
- frac12 frac14 frac34
- gt
- Iacute iacute Icirc icirc iexcl Igrave igrave iquest Iuml iuml
- laquo lt
- macr micro middot
- nbsp not Ntilde ntilde
- Oacute oacute Ocirc ocirc Ograve ograve ordf ordm Oslash oslash Otilde otilde Ouml ouml
- para permil plusmn pound
- quot
- raquo reg
- sect shy sup1 sup2 sup3 szlig
- THORN thorn times trade
- Uacute uacute Ucirc ucirc Ugrave ugrave uml Uuml uuml
- Yacute yacute
- yen Yuml yuml

31.62.2 Constructor & Destructor Documentation

Fl_Help_View::~Fl_Help_View ( )

Destroys the Fl_Help_View widget.

The destructor destroys the widget and frees all memory that has been allocated for the current document.

31.62.3 Member Function Documentation

void Fl_Help_View::clear_selection ( )

Removes the current text selection.

const char∗ Fl_Help_View::directory ( ) const `[inline]`

Returns the current directory for the text in the buffer.

void Fl_Help_View::draw (void) `[protected], [virtual]`

Draws the Fl_Help_View widget.
Reimplemented from Fl_Group.

const char∗ Fl_Help_View::filename ( ) const `[inline]`

Returns the current filename for the text in the buffer.

int Fl_Help_View::find (const char ∗ *s*, int *p* = *0*)

Finds the specified string `s` at starting position `p`.

Returns

the matching position or -1 if not found

int Fl_Help_View::handle (int *event*) `[virtual]`

Handles events in the widget.
Reimplemented from Fl_Group.

void Fl_Help_View::leftline (int *left*)

Scrolls the text to the indicated position, given a pixel column.
If the given pixel value `left` is out of range, then the text is scrolled to the left or right side of the document, resp.

Parameters

in	*left*	left column number in pixels (0 = left side)

int Fl_Help_View::leftline ( ) const `[inline]`

Gets the left position in pixels.

void Fl_Help_View::link (Fl_Help_Func ∗ *fn*) `[inline]`

This method assigns a callback function to use when a link is followed or a file is loaded (via Fl_Help_-View::load()) that requires a different file or path.
The callback function receives a pointer to the Fl_Help_View widget and the URI or full pathname for the file in question. It must return a pathname that can be opened as a local file or NULL:

```
const char *fn(Fl_Widget *w, const char *uri);
```

The link function can be used to retrieve remote or virtual documents, returning a temporary file that contains the actual data. If the link function returns NULL, the value of the Fl_Help_View widget will remain unchanged.
If the link callback cannot handle the URI scheme, it should return the uri value unchanged or set the value() of the widget before returning NULL.

int Fl_Help_View::load (const char ∗*f*)

Loads the specified file.
This method loads the specified file or URL.

void Fl_Help_View::resize (int *xx*, int *yy*, int *ww*, int *hh*) `[virtual]`

Resizes the help widget.
Reimplemented from Fl_Group.

int Fl_Help_View::scrollbar_size ( ) const `[inline]`

Gets the current size of the scrollbars' troughs, in pixels.
If this value is zero (default), this widget will use the Fl::scrollbar_size() value as the scrollbar's width.

Returns

Scrollbar size in pixels, or 0 if the global Fl::scrollbar_size() is being used.

See Also

Fl::scrollbar_size(int)

void Fl_Help_View::scrollbar_size (int *newSize*) `[inline]`

Sets the pixel size of the scrollbars' troughs to `newSize`, in pixels.
Normally you should not need this method, and should use Fl::scrollbar_size(int) instead to manage the size of ALL your widgets' scrollbars. This ensures your application has a consistent UI, is the default behavior, and is normally what you want.
Only use THIS method if you really need to override the global scrollbar size. The need for this should be rare.
Setting `newSize` to the special value of 0 causes the widget to track the global Fl::scrollbar_size(), which is the default.
Parameters

`in`	*newSize*	Sets the scrollbar size in pixels. If 0 (default), scrollbar size tracks the global Fl::scrollbar_size()

See Also

Fl::scrollbar_size()

void Fl_Help_View::select_all ( )

Selects all the text in the view.

int Fl_Help_View::size ( ) const `[inline]`

Gets the size of the help view.

void Fl_Help_View::textcolor (Fl_Color *c*) `[inline]`

Sets the default text color.

Fl_Color Fl_Help_View::textcolor ( ) const `[inline]`

Returns the current default text color.

void Fl_Help_View::textfont (Fl_Font *f*) `[inline]`

Sets the default text font.

Fl_Font Fl_Help_View::textfont ( ) const `[inline]`

Returns the current default text font.

void Fl_Help_View::textsize (Fl_Fontsize *s*) `[inline]`

Sets the default text size.

Fl_Fontsize Fl_Help_View::textsize ( ) const `[inline]`

Gets the default text size.

const char∗ Fl_Help_View::title ( ) `[inline]`

Returns the current document title, or NULL if there is no title.

void Fl_Help_View::topline (const char ∗ *n*)

Scrolls the text to the indicated position, given a named destination.
Parameters

`in`	*n*	target name

void Fl_Help_View::topline (int *top*)

Scrolls the text to the indicated position, given a pixel line.
If the given pixel value `top` is out of range, then the text is scrolled to the top or bottom of the document, resp.
Parameters

`in`	*top*	top line number in pixels (0 = start of document)

int Fl_Help_View::topline ( ) const `[inline]`

Returns the current top line in pixels.

void Fl_Help_View::value (const char ∗ *val*)

Sets the current help text buffer to the string provided and reformats the text.
The provided character string `val` is copied internally and will be freed when value() is called again, or when the widget is destroyed.
If `val` is NULL, then the widget is cleared.

const char∗ Fl_Help_View::value ( ) const `[inline]`

Returns the current buffer contents.
The documentation for this class was generated from the following files:

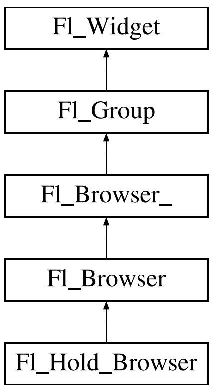
- Fl_Help_View.H
- Fl_Help_View.cxx

31.63 Fl_Hold_Browser Class Reference

The Fl_Hold_Browser is a subclass of Fl_Browser which lets the user select a single item, or no items by clicking on the empty space.

`#include <Fl_Hold_Browser.H>`

Inheritance diagram for Fl_Hold_Browser:



Public Member Functions

- Fl_Hold_Browser (int X, int Y, int W, int H, const char ∗L=0)

  Creates a new Fl_Hold_Browser widget using the given position, size, and label string.

Additional Inherited Members

31.63.1 Detailed Description

The Fl_Hold_Browser is a subclass of Fl_Browser which lets the user select a single item, or no items by clicking on the empty space.

As long as the mouse button is held down the item pointed to by it is highlighted, and this highlighting remains on when the mouse button is released. Normally the callback is done when the user releases the mouse, but you can change this with when().

See Fl_Browser for methods to add and remove lines from the browser.

31.63.2 Constructor & Destructor Documentation

Fl_Hold_Browser::Fl_Hold_Browser (int *X*, int *Y*, int *W*, int *H*, const char ∗ *L* = `0`)

Creates a new Fl_Hold_Browser widget using the given position, size, and label string.

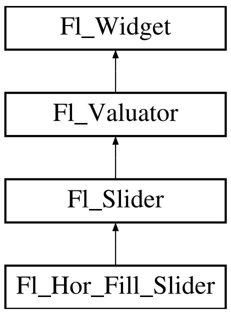
The default boxtype is FL_DOWN_BOX. The constructor specializes Fl_Browser() by setting the type to FL_HOLD_BROWSER. The destructor destroys the widget and frees all memory that has been allocated.

The documentation for this class was generated from the following files:

- Fl_Hold_Browser.H
- Fl_Browser.cxx

31.64 Fl_Hor_Fill_Slider Class Reference

Inheritance diagram for Fl_Hor_Fill_Slider:



Public Member Functions

- **Fl_Hor_Fill_Slider** (int X, int Y, int W, int H, const char ∗L=0)

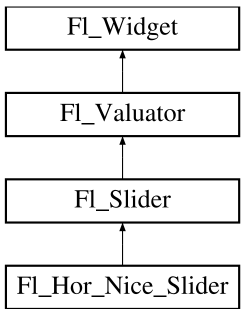
Additional Inherited Members

The documentation for this class was generated from the following files:

- Fl_Hor_Fill_Slider.H
- Fl_Slider.cxx

31.65 Fl_Hor_Nice_Slider Class Reference

Inheritance diagram for Fl_Hor_Nice_Slider:



Public Member Functions

- **Fl_Hor_Nice_Slider** (int X, int Y, int W, int H, const char ∗L=0)

Additional Inherited Members

The documentation for this class was generated from the following files:

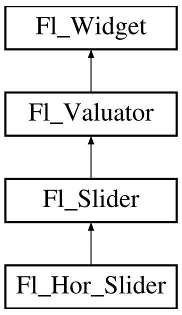
- Fl_Hor_Nice_Slider.H
- Fl_Slider.cxx

31.66 Fl_Hor_Slider Class Reference

Horizontal Slider class.

```
#include <Fl_Hor_Slider.H>
```

Inheritance diagram for Fl_Hor_Slider:



Public Member Functions

- Fl_Hor_Slider (int X, int Y, int W, int H, const char ∗l=0)

    Creates a new Fl_Hor_Slider widget using the given position, size, and label string.

Additional Inherited Members

31.66.1 Detailed Description

Horizontal Slider class.

See Also

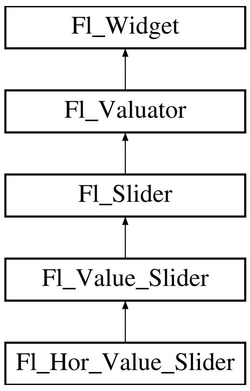
class Fl_Slider.

The documentation for this class was generated from the following files:

- Fl_Hor_Slider.H
- Fl_Slider.cxx

31.67 Fl_Hor_Value_Slider Class Reference

Inheritance diagram for Fl_Hor_Value_Slider:



Public Member Functions

- **Fl_Hor_Value_Slider** (int X, int Y, int W, int H, const char ∗l=0)

Additional Inherited Members

The documentation for this class was generated from the following files:

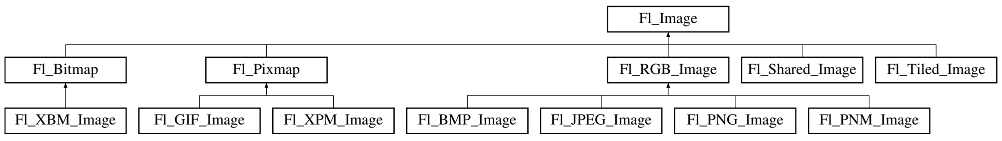
- Fl_Hor_Value_Slider.H
- Fl_Value_Slider.cxx

31.68 Fl_Image Class Reference

Base class for image caching and drawing.

`#include <Fl_Image.H>`

Inheritance diagram for Fl_Image:



Public Member Functions

- virtual void color_average (Fl_Color c, float i)

  The color_average() method averages the colors in the image with the FLTK color value c.
- virtual Fl_Image ∗ copy (int W, int H)

  The copy() method creates a copy of the specified image.
- Fl_Image ∗ copy ()

  The copy() method creates a copy of the specified image.
- int count () const

  The count() method returns the number of data values associated with the image.
- int d () const

  Returns the current image depth.
- const char ∗const ∗ data () const

  Returns a pointer to the current image data array.
- virtual void desaturate ()

  The desaturate() method converts an image to grayscale.
- virtual void draw (int X, int Y, int W, int H, int cx=0, int cy=0)

  Draws the image with a bounding box.
- void draw (int X, int Y)

  Draws the image.
- int fail ()

  Returns a value that is not 0 if there is currently no image available.
- Fl_Image (int W, int H, int D)

  The constructor creates an empty image with the specified width, height, and depth.
- int h () const

  Returns the current image height in pixels.
- void inactive ()

  The inactive() method calls color_average(FL_BACKGROUND_COLOR, 0.33f) to produce an image that appears grayed out.
- virtual void label (Fl_Widget ∗w)

The label() methods are an obsolete way to set the image attribute of a widget or menu item.

- virtual void label (Fl_Menu_Item ∗m)

  The label() methods are an obsolete way to set the image attribute of a widget or menu item.
- int ld () const

  Returns the current line data size in bytes.
- virtual void uncache ()

  If the image has been cached for display, delete the cache data.
- int w () const

  Returns the current image width in pixels.
- virtual ∼Fl_Image ()

  The destructor is a virtual method that frees all memory used by the image.

Static Public Member Functions

- static void RGB_scaling (Fl_RGB_Scaling)

  Sets the RGB image scaling method used for copy(int, int).
- static Fl_RGB_Scaling RGB_scaling ()

  Returns the currently used RGB image scaling method.

Static Public Attributes

- static const int **ERR_FILE_ACCESS** = -2
- static const int **ERR_FORMAT** = -3
- static const int **ERR_NO_IMAGE** = -1

Protected Member Functions

- void d (int D)

  Sets the current image depth.
- void data (const char ∗const ∗p, int c)

  Sets the current array pointer and count of pointers in the array.
- void draw_empty (int X, int Y)

  The protected method draw_empty() draws a box with an X in it.
- void h (int H)

  Sets the current image height in pixels.
- void ld (int LD)

  Sets the current line data size in bytes.
- void w (int W)

  Sets the current image width in pixels.

Static Protected Member Functions

- static void **labeltype** (const Fl_Label ∗lo, int lx, int ly, int lw, int lh, Fl_Align la)
- static void **measure** (const Fl_Label ∗lo, int &lw, int &lh)

31.68.1 Detailed Description

Base class for image caching and drawing.

Fl_Image is the base class used for caching and drawing all kinds of images in FLTK. This class keeps track of common image data such as the pixels, colormap, width, height, and depth. Virtual methods are used to provide type-specific image handling.

Since the Fl_Image class does not support image drawing by itself, calling the draw() method results in a box with an X in it being drawn instead.

31.68.2 Constructor & Destructor Documentation

Fl_Image::Fl_Image (int *W*, int *H*, int *D*)

The constructor creates an empty image with the specified width, height, and depth.

The width and height are in pixels. The depth is 0 for bitmaps, 1 for pixmap (colormap) images, and 1 to 4 for color images.

31.68.3 Member Function Documentation

void Fl_Image::color_average (Fl_Color *c*, float *i*) `[virtual]`

The color_average() method averages the colors in the image with the FLTK color value c.

The i argument specifies the amount of the original image to combine with the color, so a value of 1.0 results in no color blend, and a value of 0.0 results in a constant image of the specified color.

An internal copy is made of the original image before changes are applied, to avoid modifying the original image.

Reimplemented in Fl_RGB_Image, Fl_Shared_Image, Fl_Pixmap, and Fl_Tiled_Image.

Fl_Image * Fl_Image::copy (int *W*, int *H*) `[virtual]`

The copy() method creates a copy of the specified image.

If the width and height are provided, the image is resized to the specified size. The image should be deleted (or in the case of Fl_Shared_Image, released) when you are done with it.

Reimplemented in Fl_RGB_Image, Fl_Shared_Image, Fl_Pixmap, Fl_Bitmap, and Fl_Tiled_Image.

Fl_Image* Fl_Image::copy ( ) `[inline]`

The copy() method creates a copy of the specified image.

If the width and height are provided, the image is resized to the specified size. The image should be deleted (or in the case of Fl_Shared_Image, released) when you are done with it.

int Fl_Image::count ( ) const `[inline]`

The count() method returns the number of data values associated with the image.

The value will be 0 for images with no associated data, 1 for bitmap and color images, and greater than 2 for pixmap images.

int Fl_Image::d ( ) const `[inline]`

Returns the current image depth.

The return value will be 0 for bitmaps, 1 for pixmaps, and 1 to 4 for color images.

const char* const* Fl_Image::data ( ) const `[inline]`

Returns a pointer to the current image data array.

Use the count() method to find the size of the data array.

void Fl_Image::desaturate ( ) [virtual]

The desaturate() method converts an image to grayscale.

If the image contains an alpha channel (depth = 4), the alpha channel is preserved.

An internal copy is made of the original image before changes are applied, to avoid modifying the original image.

Reimplemented in Fl_RGB_Image, Fl_Shared_Image, Fl_Pixmap, and Fl_Tiled_Image.

void Fl_Image::draw (int *X*, int *Y*, int *W*, int *H*, int *cx* = *0*, int *cy* = *0*) [virtual]

Draws the image with a bounding box.

Arguments `X,Y,W,H` specify a bounding box for the image, with the origin (upper-left corner) of the image offset by the `cx` and `cy` arguments.

In other words: `fl_push_clip(X,Y,W,H)` is applied, the image is drawn with its upper-left corner at `X-cx,Y-cy` and its own width and height, `fl_pop_clip()` is applied.

Reimplemented in Fl_RGB_Image, Fl_Shared_Image, Fl_Pixmap, Fl_Bitmap, and Fl_Tiled_Image.

void Fl_Image::draw (int *X*, int *Y*) [inline]

Draws the image.

This form specifies the upper-lefthand corner of the image.

void Fl_Image::draw_empty (int *X*, int *Y*) [protected]

The protected method draw_empty() draws a box with an X in it.

It can be used to draw any image that lacks image data.

int Fl_Image::fail ( )

Returns a value that is not 0 if there is currently no image available.

Example use:

```
[..]
Fl_Box box(X,Y,W,H);
Fl_JPEG_Image jpg("/tmp/foo.jpg");
switch ( jpg.fail() ) {
    case Fl_Image::ERR_NO_IMAGE:
    case Fl_Image::ERR_FILE_ACCESS:
        fl_alert("/tmp/foo.jpg: %s", strerror(errno));    // shows actual os error to user
        exit(1);
    case Fl_Image::ERR_FORMAT:
        fl_alert("/tmp/foo.jpg: couldn't decode image");
        exit(1);
}
box.image(jpg);
[..]
```

Returns

ERR_NO_IMAGE if no image was found
ERR_FILE_ACCESS if there was a file access related error (errno should be set)
ERR_FORMAT if image decoding failed.

void Fl_Image::inactive ( ) [inline]

The inactive() method calls color_average(FL_BACKGROUND_COLOR, 0.33f) to produce an image that appears grayed out.

An internal copy is made of the original image before changes are applied, to avoid modifying the original image.

void Fl_Image::label (Fl_Widget * *widget*) `[virtual]`

The label() methods are an obsolete way to set the image attribute of a widget or menu item.

Use the image() or deimage() methods of the Fl_Widget and Fl_Menu_Item classes instead.

Reimplemented in Fl_RGB_Image, Fl_Pixmap, and Fl_Bitmap.

void Fl_Image::label (Fl_Menu_Item * *m*) `[virtual]`

The label() methods are an obsolete way to set the image attribute of a widget or menu item.

Use the image() or deimage() methods of the Fl_Widget and Fl_Menu_Item classes instead.

Reimplemented in Fl_RGB_Image, Fl_Pixmap, and Fl_Bitmap.

void Fl_Image::ld (int *LD*) `[inline]`, `[protected]`

Sets the current line data size in bytes.

Color images may contain extra data that is included after every line of color image data and is normally not present.

If `LD` is zero, then line data size is assumed to be w() * d() bytes.

If `LD` is non-zero, then it must be positive and larger than w() * d() to account for the extra data per line.

int Fl_Image::ld ( ) const `[inline]`

Returns the current line data size in bytes.

See Also

ld(int)

void Fl_Image::RGB_scaling (Fl_RGB_Scaling *method*) `[static]`

Sets the RGB image scaling method used for copy(int, int).

Applies to all RGB images, defaults to FL_RGB_SCALING_NEAREST.

Fl_RGB_Scaling Fl_Image::RGB_scaling ( ) `[static]`

Returns the currently used RGB image scaling method.

void Fl_Image::uncache ( ) `[virtual]`

If the image has been cached for display, delete the cache data.

This allows you to change the data used for the image and then redraw it without recreating an image object.

Reimplemented in Fl_RGB_Image, Fl_Shared_Image, Fl_Pixmap, and Fl_Bitmap.

The documentation for this class was generated from the following files:

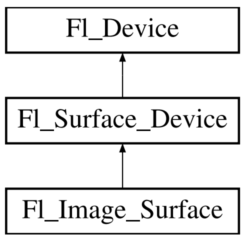
- Fl_Image.H
- Fl_Image.cxx

31.69 Fl_Image_Surface Class Reference

Directs all graphics requests to an Fl_Image.

`#include <Fl_Image_Surface.H>`

Inheritance diagram for Fl_Image_Surface:



Public Member Functions

- const char ∗ class_name ()

  Returns the name of the class of this object.
- void draw (Fl_Widget ∗, int delta_x=0, int delta_y=0)

  Draws a widget in the image surface.
- void draw_decorated_window (Fl_Window ∗win, int delta_x=0, int delta_y=0)

  Draws a window and its borders and title bar to the image drawing surface.
- Fl_Image_Surface (int w, int h, int highres=0)

  Constructor with optional high resolution.
- Fl_Shared_Image ∗ highres_image ()

  Returns a possibly high resolution image made of all drawings sent to the Fl_Image_Surface object.
- Fl_RGB_Image ∗ image ()

  Returns an image made of all drawings sent to the Fl_Image_Surface object.
- void set_current ()

  Make this surface the current drawing surface.
- ∼Fl_Image_Surface ()

  The destructor.

Static Public Attributes

- static const char ∗ **class_id** = "Fl_Image_Surface"

Additional Inherited Members

31.69.1 Detailed Description

Directs all graphics requests to an Fl_Image.

After creation of an Fl_Image_Surface object, call set_current() on it, and all subsequent graphics requests will be recorded in the image. It's possible to draw widgets (using Fl_Image_Surface::draw()) or to use any of the Drawing functions or the Color & Font functions. Finally, call image() on the object to obtain a newly allocated Fl_RGB_Image object.

Fl_GL_Window objects can be drawn in the image as well.

Usage example:

```
Fl_Widget *g = ...; // a widget you want to draw in an image
Fl_Image_Surface *img_surf = new Fl_Image_Surface(g->
      w(), g->h()); // create an Fl_Image_Surface object
img_surf->set_current(); // direct graphics requests to the image
fl_color(FL_WHITE); fl_rectf(0, 0, g->w(), g->h()); // draw a white background
img_surf->draw(g); // draw the g widget in the image
Fl_RGB_Image* image = img_surf->image(); // get the resulting image
delete img_surf; // delete the img_surf object
Fl_Display_Device::display_device()->
      set_current();  // direct graphics requests back to the display
```

31.69.2 Constructor & Destructor Documentation

Fl_Image_Surface::Fl_Image_Surface (int *w*, int *h*, int *highres* = 0)

Constructor with optional high resolution.

Parameters

w	and
h	give the size in pixels of the resulting image.
highres	if non-zero, the surface pixel size is twice as high and wide as w and h, which is useful to draw it later on a high resolution display (e.g., retina display). This is implemented for the Mac OS platform only. If `highres` is non-zero, use Fl_Image_Surface::highres_image() to get the image data.

Version

1.3.4 and requires compilation with -DFL_ABI_VERSION=10304 (1.3.3 without the highres parameter)

31.69.3 Member Function Documentation

const char* Fl_Image_Surface::class_name ( ) [inline], [virtual]

Returns the name of the class of this object.

Use of the class_name() function is discouraged because it will be removed from future FLTK versions. The class of an instance of an Fl_Device subclass can be checked with code such as:

```
if ( instance->class_name() == Fl_Printer::class_id ) { ... }
```

Reimplemented from Fl_Surface_Device.

void Fl_Image_Surface::draw (Fl_Widget * *widget*, int *delta_x* = 0, int *delta_y* = 0)

Draws a widget in the image surface.

Parameters

widget	any FLTK widget (e.g., standard, custom, window, GL view) to draw in the image
delta_x	and
delta_y	give the position in the image of the top-left corner of the widget

void Fl_Image_Surface::draw_decorated_window (Fl_Window * *win*, int *delta_x* = 0, int *delta_y* = 0)

Draws a window and its borders and title bar to the image drawing surface.

Parameters

win	an FLTK window to draw in the image
delta_x	and
delta_y	give the position in the image of the top-left corner of the window's title bar

Fl_Shared_Image ∗ Fl_Image_Surface::highres_image ( )

Returns a possibly high resolution image made of all drawings sent to the Fl_Image_Surface object.

The Fl_Image_Surface object should have been constructed with Fl_Image_Surface(W, H, 1). The returned image is scaled to a size of WxH drawing units and may have a pixel size twice as wide and high. The returned object should be deallocated with Fl_Shared_Image::release() after use.

Version

1.3.4 and requires compilation with -DFL_ABI_VERSION=10304

Fl_RGB_Image ∗ Fl_Image_Surface::image ( )

Returns an image made of all drawings sent to the Fl_Image_Surface object.

The returned object contains its own copy of the RGB data. Prefer Fl_Image_Surface::highres_image() if the surface was constructed with the highres option on.

void Fl_Image_Surface::set_current (void) `[virtual]`

Make this surface the current drawing surface.

This surface will receive all future graphics requests.

Reimplemented from Fl_Surface_Device.

The documentation for this class was generated from the following files:

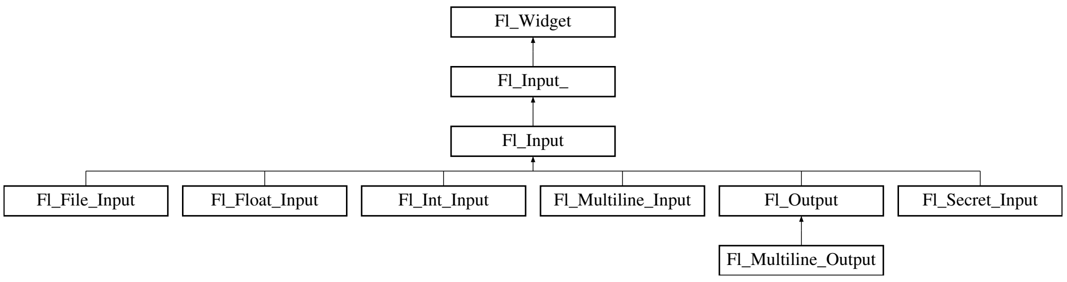
- Fl_Image_Surface.H
- Fl_Image_Surface.cxx

31.70 Fl_Input Class Reference

This is the FLTK text input widget.

`#include <Fl_Input.H>`

Inheritance diagram for Fl_Input:



Public Member Functions

- Fl_Input (int, int, int, int, const char ∗=0)

  Creates a new Fl_Input widget using the given position, size, and label string.
- int handle (int)

  Handles the specified event.

Protected Member Functions

- void draw ()

  Draws the widget.

Additional Inherited Members

31.70.1 Detailed Description

This is the FLTK text input widget.

It displays a single line of text and lets the user edit it. Normally it is drawn with an inset box and a white background. The text may contain any characters, and will correctly display any UTF text, using ^X notation for unprintable control characters. It assumes the font can draw any characters of the used scripts, which is true for standard fonts under MSWindows and Mac OS X. Characters can be input using the keyboard or the character palette/map. Character composition is done using dead keys and/or a compose key as defined by the operating system.

Mouse button 1	Moves the cursor to this point. Drag selects characters. Double click selects words. Triple click selects all line. Shift+click extends the selection. When you select text it is automatically copied to the selection buffer.
Mouse button 2	Insert the selection buffer at the point clicked. You can also select a region and replace it with the selection buffer by selecting the region with mouse button 2.
Mouse button 3	Currently acts like button 1.
Backspace	Deletes one character to the left, or deletes the selected region.
Delete	Deletes one character to the right, or deletes the selected region. Combine with Shift for equivalent of ^X (copy+cut).
Enter	May cause the callback, see when().

Table 31.1: Fl_Input keyboard and mouse bindings.

31.70.2 Constructor & Destructor Documentation

Fl_Input::Fl_Input (int *X*, int *Y*, int *W*, int *H*, const char ∗ *l* = 0)

Creates a new Fl_Input widget using the given position, size, and label string.

The default boxtype is FL_DOWN_BOX.

31.70.3 Member Function Documentation

void Fl_Input::draw ( ) `[protected]`, `[virtual]`

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call redraw() instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw() method*, e.g. for an embedded scrollbar, you can do it (because draw() is virtual) like this:

```
Fl_Widget *s = &scroll;              // scroll is an embedded Fl_Scrollbar
s->draw();                    // calls Fl_Scrollbar::draw()
```

Implements Fl_Widget.

int Fl_Input::handle (int *event*) `[virtual]`

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited handle() method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters

`in`	*event*	the kind of event received

Return values

0	if the event was not used or understood
1	if the event was used and can be deleted

See Also

Fl_Event

Reimplemented from Fl_Widget.

Reimplemented in Fl_Secret_Input.

The documentation for this class was generated from the following files:

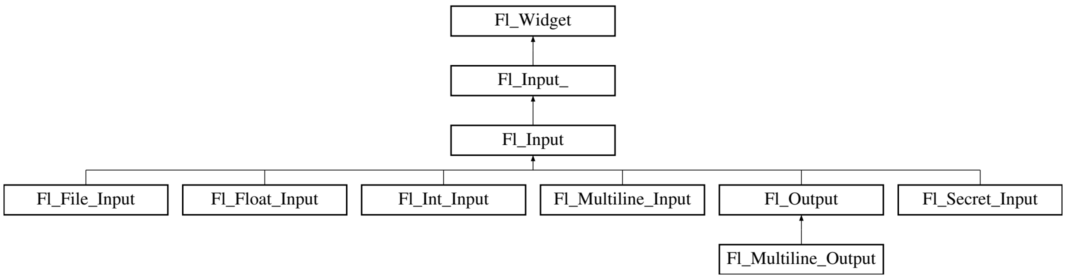
- Fl_Input.H
- Fl_Input.cxx

31.71 Fl_Input_ Class Reference

This class provides a low-overhead text input field.

`#include <Fl_Input_.H>`

Inheritance diagram for Fl_Input_:



Public Member Functions

- int copy (int clipboard)

  Put the current selection into the clipboard.
- int copy_cuts ()

  Copies the yank buffer to the clipboard.
- Fl_Color cursor_color () const

  Gets the color of the cursor.
- void cursor_color (Fl_Color n)

  Sets the color of the cursor.
- int cut ()

  Deletes the current selection.
- int cut (int n)

  Deletes the next `n` bytes rounded to characters before or after the cursor.
- int cut (int a, int b)

  Deletes all characters between index `a` and `b`.
- Fl_Input_ (int, int, int, int, const char ∗=0)

  Creates a new Fl_Input_ widget.
- Fl_Char index (int i) const

  Returns the character at index `i`.
- int input_type () const

  Gets the input field type.
- void input_type (int t)

  Sets the input field type.
- int insert (const char ∗t, int l=0)

  Inserts text at the cursor position.
- int mark () const

  Gets the current selection mark.
- int mark (int m)

  Sets the current selection mark.
- int maximum_size () const

  Gets the maximum length of the input field in characters.
- void maximum_size (int m)

  Sets the maximum length of the input field in characters.
- int position () const

  Gets the position of the text cursor.
- int position (int p, int m)

  Sets the index for the cursor and mark.

- int position (int p)

    Sets the cursor position and mark.
- int readonly () const

    Gets the read-only state of the input field.
- void readonly (int b)

    Sets the read-only state of the input field.
- int replace (int b, int e, const char ∗text, int ilen=0)

    Deletes text from `b` *to* `e` *and inserts the new string* `text`*.*
- void resize (int, int, int, int)

    Changes the size of the widget.
- int shortcut () const

    Return the shortcut key associated with this widget.
- void shortcut (int s)

    Sets the shortcut key associated with this widget.
- int size () const

    Returns the number of bytes in value().
- void size (int W, int H)

    Sets the width and height of this widget.
- int static_value (const char ∗)

    Changes the widget text.
- int static_value (const char ∗, int)

    Changes the widget text.
- void tab_nav (int val)

    Sets whether the Tab key does focus navigation, or inserts tab characters into Fl_Multiline_Input.
- int tab_nav () const

    Gets whether the Tab key causes focus navigation in multiline input fields or not.
- Fl_Color textcolor () const

    Gets the color of the text in the input field.
- void textcolor (Fl_Color n)

    Sets the color of the text in the input field.
- Fl_Font textfont () const

    Gets the font of the text in the input field.
- void textfont (Fl_Font s)

    Sets the font of the text in the input field.
- Fl_Fontsize textsize () const

    Gets the size of the text in the input field.
- void textsize (Fl_Fontsize s)

    Sets the size of the text in the input field.
- int undo ()

    Undoes previous changes to the text buffer.
- int value (const char ∗)

    Changes the widget text.
- int value (const char ∗, int)

    Changes the widget text.
- const char ∗ value () const

    Returns the text displayed in the widget.

- int wrap () const

  Gets the word wrapping state of the input field.
- void wrap (int b)

  Sets the word wrapping state of the input field.
- ∼Fl_Input_ ()

  Destroys the widget.

Protected Member Functions

- void drawtext (int, int, int, int)

  Draws the text in the passed bounding box.
- void handle_mouse (int, int, int, int, int keepmark=0)

  Handles mouse clicks and mouse moves.
- int handletext (int e, int, int, int, int)

  Handles all kinds of text field related events.
- int line_end (int i) const

  Finds the end of a line.
- int line_start (int i) const

  Finds the start of a line.
- int **linesPerPage** ()
- void **maybe_do_callback** ()
- int up_down_position (int, int keepmark=0)

  Moves the cursor to the column given by `up_down_pos`*.*
- int word_end (int i) const

  Finds the end of a word.
- int word_start (int i) const

  Finds the start of a word.
- int **xscroll** () const
- int **yscroll** () const
- void **yscroll** (int yOffset)

Additional Inherited Members

31.71.1 Detailed Description

This class provides a low-overhead text input field.

This is a virtual base class below Fl_Input. It has all the same interfaces, but lacks the handle() and draw() method. You may want to subclass it if you are one of those people who likes to change how the editing keys work. It may also be useful for adding scrollbars to the input field.

This can act like any of the subclasses of Fl_Input, by setting type() to one of the following values:

```
#define FL_NORMAL_INPUT     0
#define FL_FLOAT_INPUT      1
#define FL_INT_INPUT                2
#define FL_MULTILINE_INPUT          4
#define FL_SECRET_INPUT     5
#define FL_INPUT_TYPE               7
#define FL_INPUT_READONLY           8
#define FL_NORMAL_OUTPUT            (FL_NORMAL_INPUT | FL_INPUT_READONLY)
#define FL_MULTILINE_OUTPUT         (FL_MULTILINE_INPUT | FL_INPUT_READONLY)
#define FL_INPUT_WRAP               16
#define FL_MULTILINE_INPUT_WRAP   (FL_MULTILINE_INPUT | FL_INPUT_WRAP)
#define FL_MULTILINE_OUTPUT_WRAP  (FL_MULTILINE_INPUT | FL_INPUT_READONLY | FL_INPUT_WRAP)
```

All variables that represent an index into a text buffer are byte-oriented, not character oriented, counting from 0 (at or before the first character) to size() (at the end of the buffer, after the last byte). Since UTF-8 characters can be up to six bytes long, simply incrementing such an index will not reliably advance to the next character in the text buffer.

Indices and pointers into the text buffer should always point at a 7 bit ASCII character or the beginning of a UTF-8 character sequence. Behavior for false UTF-8 sequences and pointers into the middle of a sequence are undefined.

See Also

Fl_Text_Display, Fl_Text_Editor for more powerful text handling widgets

31.71.2 Constructor & Destructor Documentation

Fl_Input_::Fl_Input_ (int *X*, int *Y*, int *W*, int *H*, const char ∗ *l* = `0`)

Creates a new Fl_Input_ widget.

This function creates a new Fl_Input_ widget and adds it to the current Fl_Group. The value() is set to `NULL`. The default boxtype is `FL_DOWN_BOX`.

Parameters

X,Y,W,H	the dimensions of the new widget
l	an optional label text

Fl_Input_::∼Fl_Input_ ( )

Destroys the widget.

The destructor clears all allocated buffers and removes the widget from the parent Fl_Group.

31.71.3 Member Function Documentation

int Fl_Input_::copy (int *clipboard*)

Put the current selection into the clipboard.

This function copies the current selection between mark() and position() into the specified `clipboard`. This does not replace the old clipboard contents if position() and mark() are equal. Clipboard 0 maps to the current text selection and clipboard 1 maps to the cut/paste clipboard.

Parameters

clipboard	the clipboard destination 0 or 1

Returns

0 if no text is selected, 1 if the selection was copied

See Also

Fl::copy(const char ∗, int, int)

int Fl_Input_::copy_cuts ( )

Copies the *yank* buffer to the clipboard.

This method copies all the previous contiguous cuts from the undo information to the clipboard. This function implements the ^`K` shortcut key.

Returns

0 if the operation did not change the clipboard

See Also

copy(int), cut()

Fl_Color Fl_Input_::cursor_color ( ) const `[inline]`

Gets the color of the cursor.

Returns

the current cursor color

void Fl_Input_::cursor_color (Fl_Color *n*) `[inline]`

Sets the color of the cursor.

The default color for the cursor is `FL_BLACK`.

Parameters

`in`	*n*	the new cursor color

int Fl_Input_::cut ( ) `[inline]`

Deletes the current selection.

This function deletes the currently selected text *without* storing it in the clipboard. To use the clipboard, you may call copy() first or copy_cuts() after this call.

Returns

0 if no data was copied

int Fl_Input_::cut (int *n*) `[inline]`

Deletes the next `n` bytes rounded to characters before or after the cursor.

This function deletes the currently selected text *without* storing it in the clipboard. To use the clipboard, you may call copy() first or copy_cuts() after this call.

Parameters

n	number of bytes rounded to full characters and clamped to the buffer. A negative number will cut characters to the left of the cursor.

Returns

0 if no data was copied

int Fl_Input_::cut (int *a*, int *b*) `[inline]`

Deletes all characters between index `a` and `b`.

This function deletes the currently selected text *without* storing it in the clipboard. To use the clipboard, you may call copy() first or copy_cuts() after this call.

Parameters

a,b	range of bytes rounded to full characters and clamped to the buffer

Returns

0 if no data was copied

void Fl_Input_::drawtext (int *X*, int *Y*, int *W*, int *H*) `[protected]`

Draws the text in the passed bounding box.

If `damage() & FL_DAMAGE_ALL` is true, this assumes the area has already been erased to color(). Otherwise it does minimal update and erases the area itself.

Parameters

X,Y,W,H	area that must be redrawn

void Fl_Input_::handle_mouse (int *X*, int *Y*, int , int , int *drag* = *0*) `[protected]`

Handles mouse clicks and mouse moves.

Todo Add comment and parameters

int Fl_Input_::handletext (int *event*, int *X*, int *Y*, int *W*, int *H*) `[protected]`

Handles all kinds of text field related events.

This is called by derived classes.

Todo Add comment and parameters

unsigned int Fl_Input_::index (int *i*) const

Returns the character at index `i`.

This function returns the UTF-8 character at `i` as a ucs4 character code.

Parameters

`in`	*i*	index into the value field

Returns

the character at index `i`

int Fl_Input_::input_type ( ) const `[inline]`

Gets the input field type.

Returns

the current input type

void Fl_Input_::input_type (int *t*) `[inline]`

Sets the input field type.

A redraw() is required to reformat the input field.

Parameters

in	*t*	new input type

int Fl_Input_::insert (const char * *t*, int *l* = *0*) `[inline]`

Inserts text at the cursor position.

This function inserts the string in `t` at the cursor position() and moves the new position and mark to the end of the inserted text.

Parameters

in	*t*	text that will be inserted
in	*l*	length of text, or 0 if the string is terminated by `nul`.

Returns

0 if no text was inserted

int Fl_Input_::line_end (int *i*) const `[protected]`

Finds the end of a line.

This call calculates the end of a line based on the given index `i`.

Parameters

in	*i*	starting index for the search

Returns

end of the line

int Fl_Input_::line_start (int *i*) const `[protected]`

Finds the start of a line.

This call calculates the start of a line based on the given index `i`.

Parameters

in	*i*	starting index for the search

Returns

start of the line

int Fl_Input_::mark ( ) const `[inline]`

Gets the current selection mark.

Returns

index into the text

int Fl_Input_::mark (int *m*) `[inline]`

Sets the current selection mark.

mark(n) is the same as `position(position(),n)`.

Parameters

m	new index of the mark

Returns

0 if the mark did not change

See Also

position(), position(int, int)

int Fl_Input_::maximum_size ( ) const `[inline]`

Gets the maximum length of the input field in characters.

See Also

maximum_size(int).

void Fl_Input_::maximum_size (int *m*) `[inline]`

Sets the maximum length of the input field in characters.

This limits the number of **characters** that can be inserted in the widget.

Since FLTK 1.3 this is different than the buffer size, since one character can be more than one byte in UTF-8 encoding. In FLTK 1.1 this was the same (one byte = one character).

int Fl_Input_::position ( ) const `[inline]`

Gets the position of the text cursor.

Returns

the cursor position as an index in the range 0..size()

See Also

position(int, int)

int Fl_Input_::position (int *p*, int *m*)

Sets the index for the cursor and mark.

The input widget maintains two pointers into the string. The *position* (`p`) is where the cursor is. The *mark* (`m`) is the other end of the selected text. If they are equal then there is no selection. Changing this does not affect the clipboard (use copy() to do that).

Changing these values causes a redraw(). The new values are bounds checked.

Parameters

p	index for the cursor position
m	index for the mark

Returns

0 if no positions changed

See Also

position(int), position(), mark(int)

int Fl_Input_::position (int *p*) `[inline]`

Sets the cursor position and mark.

position(n) is the same as `position(n, n)`.

Parameters

p	new index for cursor and mark

Returns

0 if no positions changed

See Also

position(int, int), position(), mark(int)

int Fl_Input_::readonly ( ) const `[inline]`

Gets the read-only state of the input field.

Returns

non-zero if this widget is read-only

void Fl_Input_::readonly (int *b*) `[inline]`

Sets the read-only state of the input field.

Parameters

`in`	*b*	if `b` is 0, the text in this widget can be edited by the user

int Fl_Input_::replace (int *b,* int *e,* const char ∗ *text,* int *ilen* = `0`)

Deletes text from `b` to `e` and inserts the new string `text`.

All changes to the text buffer go through this function. It deletes the region between `b` and `e` (either one may be less or equal to the other), and then inserts the string `text` at that point and moves the mark() and position() to the end of the insertion. Does the callback if `when() & FL_WHEN_CHANGED` and there is a change.

Set `b` and `e` equal to not delete anything. Set `text` to `NULL` to not insert anything.

`ilen` can be zero or `strlen(text)`, which saves a tiny bit of time if you happen to already know the length of the insertion, or can be used to insert a portion of a string. If `ilen` is zero, `strlen(text)` is used instead.

`b` and `e` are clamped to the `0..size()` range, so it is safe to pass any values. `b`, `e`, and `ilen` are used as numbers of bytes (not characters), where `b` and `e` count from 0 to size() (end of buffer).

If `b` and/or `e` don't point to a valid UTF-8 character boundary, they are adjusted to the previous (`b`) or the next (`e`) valid UTF-8 character boundary, resp..

If the current number of characters in the buffer minus deleted characters plus inserted characters in `text` would overflow the number of allowed characters (maximum_size()), then only the first characters of the string are inserted, so that maximum_size() is not exceeded.

cut() and insert() are just inline functions that call replace().

Parameters

`in`	*b*	beginning index of text to be deleted
`in`	*e*	ending index of text to be deleted and insertion position
`in`	*text*	string that will be inserted
`in`	*ilen*	length of `text` or 0 for `nul` terminated strings

Returns

0 if nothing changed

Note

If `text` does not point to a valid UTF-8 character or includes invalid UTF-8 sequences, the text is inserted nevertheless (counting invalid UTF-8 bytes as one character each).

void Fl_Input_::resize (int *X*, int *Y*, int *W*, int *H*) `[virtual]`

Changes the size of the widget.

This call updates the text layout so that the cursor is visible.

Parameters

`in`	*X,Y,W,H*	new size of the widget

See Also

Fl_Widget::resize(int, int, int, int)

Reimplemented from Fl_Widget.

int Fl_Input_::shortcut ( ) const `[inline]`

Return the shortcut key associated with this widget.

Returns

shortcut keystroke

See Also

Fl_Button::shortcut()

void Fl_Input_::shortcut (int *s*) `[inline]`

Sets the shortcut key associated with this widget.

Pressing the shortcut key gives text editing focus to this widget.

Parameters

`in`	*s*	new shortcut keystroke

See Also

Fl_Button::shortcut()

int Fl_Input_::size ( ) const `[inline]`

Returns the number of bytes in value().

This may be greater than `strlen(value())` if there are `nul` characters in the text.

Returns

number of bytes in the text

void Fl_Input_::size (int *W*, int *H*) `[inline]`

Sets the width and height of this widget.

Parameters

`in`	*W,H*	new width and height

See Also

Fl_Widget::size(int, int)

int Fl_Input_::static_value (const char ∗ *str*)

Changes the widget text.

This function changes the text and sets the mark and the point to the end of it. The string is *not* copied. If the user edits the string it is copied to the internal buffer then. This can save a great deal of time and memory if your program is rapidly changing the values of text fields, but this will only work if the passed string remains unchanged until either the Fl_Input is destroyed or value() is called again.

Parameters

`in`	*str*	the new text

Returns

non-zero if the new value is different than the current one

int Fl_Input_::static_value (const char ∗ *str*, int *len*)

Changes the widget text.

This function changes the text and sets the mark and the point to the end of it. The string is *not* copied. If the user edits the string it is copied to the internal buffer then. This can save a great deal of time and memory if your program is rapidly changing the values of text fields, but this will only work if the passed string remains unchanged until either the Fl_Input is destroyed or value() is called again.

You can use the `len` parameter to directly set the length if you know it already or want to put `nul` characters in the text.

Parameters

`in`	*str*	the new text
`in`	*len*	the length of the new text

Returns

non-zero if the new value is different than the current one

void Fl_Input_::tab_nav (int *val*) `[inline]`

Sets whether the Tab key does focus navigation, or inserts tab characters into Fl_Multiline_Input.

By default this flag is enabled to provide the 'normal' behavior most users expect; Tab navigates focus to the next widget. To inserting an actual Tab character, users can use Ctrl-I or copy/paste.

Disabling this flag gives the old FLTK behavior where Tab inserts a tab character into the text field, in which case only the mouse can be used to navigate to the next field.

History: This flag was provided for backwards support of FLTK's old 1.1.x behavior where Tab inserts a tab character instead of navigating focus to the next widget. This behavior was unique to Fl_Multiline_Input. With the advent of Fl_Text_Editor, this old behavior has been deprecated.

Parameters

`in`	*val*	If `val` is 1, Tab advances focus (default). If `val` is 0, Tab inserts a tab character (old FLTK behavior).

See Also

tab_nav(), Fl::OPTION_ARROW_FOCUS.

int Fl_Input_::tab_nav ( ) const `[inline]`

Gets whether the Tab key causes focus navigation in multiline input fields or not.

If enabled (default), hitting Tab causes focus navigation to the next widget.

If disabled, hitting Tab inserts a tab character into the text field.

Returns

1 if Tab advances focus (default), 0 if Tab inserts tab characters.

See Also

tab_nav(int), Fl::OPTION_ARROW_FOCUS.

Fl_Color Fl_Input_::textcolor ( ) const `[inline]`

Gets the color of the text in the input field.

Returns

the text color

See Also

textcolor(Fl_Color)

void Fl_Input_::textcolor (Fl_Color *n*) `[inline]`

Sets the color of the text in the input field.

The text color defaults to `FL_FOREGROUND_COLOR`.

Parameters

`in`	*n*	new text color

See Also

textcolor()

Fl_Font Fl_Input_::textfont ( ) const `[inline]`

Gets the font of the text in the input field.

Returns

the current Fl_Font index

void Fl_Input_::textfont (Fl_Font ***s*** **)** `[inline]`

Sets the font of the text in the input field.
The text font defaults to `FL_HELVETICA`.
Parameters

`in`	*s*	the new text font

Fl_Fontsize Fl_Input_::textsize ( ) const `[inline]`

Gets the size of the text in the input field.

Returns

the text height in pixels

void Fl_Input_::textsize (Fl_Fontsize ***s*** **)** `[inline]`

Sets the size of the text in the input field.
The text height defaults to `FL_NORMAL_SIZE`.
Parameters

`in`	*s*	the new font height in pixel units

int Fl_Input_::undo ( )

Undoes previous changes to the text buffer.
This call undoes a number of previous calls to replace().

Returns

non-zero if any change was made.

int Fl_Input_::up_down_position (int ***i*****, int** ***keepmark*** **=** `0` **)** `[protected]`

Moves the cursor to the column given by `up_down_pos`.
This function is helpful when implementing up and down cursor movement. It moves the cursor from the beginning of a line to the column indicated by the global variable `up_down_pos` in pixel units.
Parameters

`in`	*i*	index into the beginning of a line of text
`in`	*keepmark*	if set, move only the cursor, but not the mark

Returns

index to new cursor position

int Fl_Input_::value (const char * *str*)

Changes the widget text.

This function changes the text and sets the mark and the point to the end of it. The string is copied to the internal buffer. Passing `NULL` is the same as `""`.

Parameters

`in`	*str*	the new text

Returns

non-zero if the new value is different than the current one

See Also

Fl_Input_::value(const char* str, int len), Fl_Input_::value()

int Fl_Input_::value (const char * *str,* int *len*)

Changes the widget text.

This function changes the text and sets the mark and the point to the end of it. The string is copied to the internal buffer. Passing `NULL` is the same as "".

You can use the `length` parameter to directly set the length if you know it already or want to put `nul` characters in the text.

Parameters

`in`	*str*	the new text
`in`	*len*	the length of the new text

Returns

non-zero if the new value is different than the current one

See Also

Fl_Input_::value(const char* str), Fl_Input_::value()

const char* Fl_Input_::value ( ) const `[inline]`

Returns the text displayed in the widget.

This function returns the current value, which is a pointer to the internal buffer and is valid only until the next event is handled.

Returns

pointer to an internal buffer - do not free() this

See Also

Fl_Input_::value(const char*)

int Fl_Input_::word_end (int *i*) const `[protected]`

Finds the end of a word.

Returns the index after the last byte of a word. If the index is already at the end of a word, it will find the end of the following word, so if you call it repeatedly you will move forwards to the end of the text.

Note that this is inconsistent with line_end().

Parameters

in	*i*	starting index for the search

Returns

end of the word

int Fl_Input_::word_start (int *i*) const [protected]

Finds the start of a word.

Returns the index of the first byte of a word. If the index is already at the beginning of a word, it will find the beginning of the previous word, so if you call it repeatedly you will move backwards to the beginning of the text.

Note that this is inconsistent with line_start().

Parameters

in	*i*	starting index for the search

Returns

start of the word, or previous word

int Fl_Input_::wrap ( ) const [inline]

Gets the word wrapping state of the input field.

Word wrap is only functional with multi-line input fields.

void Fl_Input_::wrap (int *b*) [inline]

Sets the word wrapping state of the input field.

Word wrap is only functional with multi-line input fields.

The documentation for this class was generated from the following files:

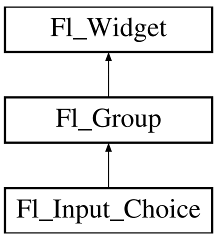
- Fl_Input_.H
- Fl_Input_.cxx

31.72 Fl_Input_Choice Class Reference

A combination of the input widget and a menu button.

`#include <Fl_Input_Choice.H>`

Inheritance diagram for Fl_Input_Choice:



Public Member Functions

- void add (const char ∗s)

  Adds an item to the menu.
- int changed () const

  Returns the combined changed() state of the input and menu button widget.
- void clear ()

  Removes all items from the menu.
- void clear_changed ()

  Clears the changed() state of both input and menu button widgets.
- Fl_Boxtype down_box () const

  Gets the box type of the menu button.
- void down_box (Fl_Boxtype b)

  Sets the box type of the menu button.
- Fl_Input_Choice (int X, int Y, int W, int H, const char ∗L=0)

  Creates a new Fl_Input_Choice widget using the given position, size, and label string.
- Fl_Input ∗ input ()

  Returns a pointer to the internal Fl_Input widget.
- const Fl_Menu_Item ∗ menu ()

  Gets the Fl_Menu_Item array used for the menu.
- void menu (const Fl_Menu_Item ∗m)

  Sets the Fl_Menu_Item array used for the menu.
- Fl_Menu_Button ∗ menubutton ()

  Returns a pointer to the internal Fl_Menu_Button widget.
- void resize (int X, int Y, int W, int H)

  Resizes the Fl_Group widget and all of its children.
- void set_changed ()

  Sets the changed() state of both input and menu button widgets to the specfied value.
- Fl_Color textcolor () const

  Gets the Fl_Input text field's text color.
- void textcolor (Fl_Color c)

  Sets the Fl_Input text field's text color to `c`*.*
- Fl_Font textfont () const

  Gets the Fl_Input text field's font style.
- void textfont (Fl_Font f)

  Sets the Fl_Input text field's font style to `f`*.*
- Fl_Fontsize textsize () const

  Gets the Fl_Input text field's font size.
- void textsize (Fl_Fontsize s)

  Sets the Fl_Input text field's font size to `s`*.*
- const char ∗ value () const

  Returns the Fl_Input text field's current contents.
- void value (const char ∗val)

  Sets the Fl_Input text field's contents to `val`*.*
- void value (int val)

  Chooses item# `val` *in the menu, and sets the Fl_Input text field to that value.*

Additional Inherited Members

31.72.1 Detailed Description

A combination of the input widget and a menu button.

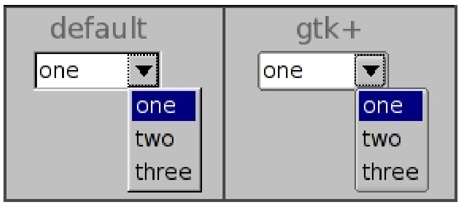


Figure 31.17: Fl_Input_Choice widget

The user can either type into the input area, or use the menu button chooser on the right to choose an item which loads the input area with the selected text.

The application can directly access both the internal Fl_Input and Fl_Menu_Button widgets respectively using the input() and menubutton() accessor methods.

The default behavior is to invoke the Fl_Input_Choice::callback() if the user changes the input field's contents, either by typing, pasting, or clicking a different item in the choice menu.

The callback can determine if an item was picked vs. typing into the input field by checking the value of menubutton()->changed(), which will be:

```
- 1: the user picked a different item in the choice menu
- 0: the user typed or pasted directly into the input field
```

Example use:

```
#include <stdio.h>
#include <FL/Fl.H>
#include <FL/Fl_Double_Window.H>
#include <FL/Fl_Input_Choice.H>
void choice_cb(Fl_Widget *w, void *userdata) {
  // Show info about the picked item
  Fl_Input_Choice *choice = (Fl_Input_Choice*)w;
  const Fl_Menu_Item *item = choice->menubutton()->mvalue();
  printf("*** Choice Callback:\n");
  printf("    item label()='%s'\n", item ? item->label() : "(No item)");
  printf("    item value()=%d\n", choice->menubutton()->value());
  printf("    input value()='%s'\n", choice->input()->value());
  printf("    The user %s\n", choice->menubutton()->changed()
                               ? "picked a menu item"
                               : "typed text");
}
int main() {
  Fl_Double_Window win(200,100,"Input Choice");
  win.begin();
    Fl_Input_Choice choice(10,10,100,30);
    choice.callback(choice_cb, 0);
    choice.add("Red");
    choice.add("Orange");
    choice.add("Yellow");
    //choice.value("Red");    // uncomment to make "Red" default
  win.end();
  win.show();
  return Fl::run();
}
```

31.72.2 Constructor & Destructor Documentation

Fl_Input_Choice::Fl_Input_Choice (int *X*, int *Y*, int *W*, int *H*, const char ∗ *L* = *0*)

Creates a new Fl_Input_Choice widget using the given position, size, and label string.

Inherited destructor destroys the widget and any values associated with it.

31.72.3 Member Function Documentation

void Fl_Input_Choice::add (const char ∗ *s*) `[inline]`

Adds an item to the menu.

You can access the more complex Fl_Menu_Button::add() methods (setting callbacks, userdata, etc), via menubutton(). Example:

```
Fl_Input_Choice *choice = new Fl_Input_Choice(100,10,120,25,"Fonts");
Fl_Menu_Button *mb = choice->menubutton();             // use Fl_Input_Choice's
       Fl_Menu_Button
mb->add("Helvetica", 0, MyFont_CB,     (void*)mydata); // use Fl_Menu_Button's add() methods
mb->add("Courier",   0, MyFont_CB,     (void*)mydata);
mb->add("More..",    0, FontDialog_CB, (void*)mydata);
```

int Fl_Input_Choice::changed ( ) const `[inline]`

Returns the combined changed() state of the input and menu button widget.

void Fl_Input_Choice::clear ( ) `[inline]`

Removes all items from the menu.

void Fl_Input_Choice::clear_changed ( ) `[inline]`

Clears the changed() state of both input and menu button widgets.

Fl_Input∗ Fl_Input_Choice::input ( ) `[inline]`

Returns a pointer to the internal Fl_Input widget.

This can be used to directly access all of the Fl_Input widget's methods.

const Fl_Menu_Item∗ Fl_Input_Choice::menu ( ) `[inline]`

Gets the Fl_Menu_Item array used for the menu.

void Fl_Input_Choice::menu (const Fl_Menu_Item ∗ *m*) `[inline]`

Sets the Fl_Menu_Item array used for the menu.

Fl_Menu_Button∗ Fl_Input_Choice::menubutton ( ) `[inline]`

Returns a pointer to the internal Fl_Menu_Button widget.

This can be used to access any of the methods of the menu button, e.g.

```
Fl_Input_Choice *choice = new Fl_Input_Choice(100,10,120,25,"Choice:");
[..]
// Print all the items in the choice menu
for ( int t=0; t<choice->menubutton()->size(); t++ ) {
    const Fl_Menu_Item &item = choice->menubutton()->menu()[t];
    printf("item %d -- label=%s\n", t, item.label() ? item.label() : "(Null)");
}
```

void Fl_Input_Choice::resize (int *X*, int *Y*, int *W*, int *H*) `[inline]`, `[virtual]`

Resizes the Fl_Group widget and all of its children.

The Fl_Group widget first resizes itself, and then it moves and resizes all its children according to the rules documented for Fl_Group::resizable(Fl_Widget∗)

See Also

Fl_Group::resizable(Fl_Widget∗)
Fl_Group::resizable()
Fl_Widget::resize(int,int,int,int)

Reimplemented from Fl_Group.

void Fl_Input_Choice::set_changed ( ) `[inline]`

Sets the changed() state of both input and menu button widgets to the specfied value.

void Fl_Input_Choice::value (const char ∗ *val*) `[inline]`

Sets the Fl_Input text field's contents to `val`.

Does not affect the menu selection.

void Fl_Input_Choice::value (int *val*) `[inline]`

Chooses item# `val` in the menu, and sets the Fl_Input text field to that value.

Any previous text is cleared.

The documentation for this class was generated from the following files:

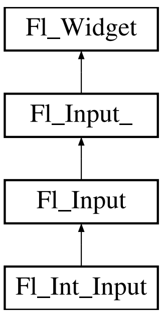
- Fl_Input_Choice.H
- Fl_Group.cxx

31.73 Fl_Int_Input Class Reference

The Fl_Int_Input class is a subclass of Fl_Input that only allows the user to type decimal digits (or hex numbers of the form 0xaef).

`#include <Fl_Int_Input.H>`

Inheritance diagram for Fl_Int_Input:



Public Member Functions

- Fl_Int_Input (int X, int Y, int W, int H, const char ∗l=0)

  Creates a new Fl_Int_Input widget using the given position, size, and label string.

Additional Inherited Members

31.73.1 Detailed Description

The Fl_Int_Input class is a subclass of Fl_Input that only allows the user to type decimal digits (or hex numbers of the form 0xaef).

31.73.2 Constructor & Destructor Documentation

Fl_Int_Input::Fl_Int_Input (int *X*, int *Y*, int *W*, int *H*, const char ∗ *l* = *0*)

Creates a new Fl_Int_Input widget using the given position, size, and label string.
The default boxtype is FL_DOWN_BOX.
Inherited destructor destroys the widget and any value associated with it.
The documentation for this class was generated from the following files:

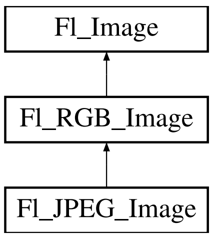
- Fl_Int_Input.H
- Fl_Input.cxx

31.74 Fl_JPEG_Image Class Reference

The Fl_JPEG_Image class supports loading, caching, and drawing of Joint Photographic Experts Group (JPEG) File Interchange Format (JFIF) images.

```
#include <Fl_JPEG_Image.H>
```

Inheritance diagram for Fl_JPEG_Image:



Public Member Functions

- Fl_JPEG_Image (const char ∗filename)

  The constructor loads the JPEG image from the given jpeg filename.
- Fl_JPEG_Image (const char ∗name, const unsigned char ∗data)

  The constructor loads the JPEG image from memory.

Additional Inherited Members

31.74.1 Detailed Description

The Fl_JPEG_Image class supports loading, caching, and drawing of Joint Photographic Experts Group (JPEG) File Interchange Format (JFIF) images.
The class supports grayscale and color (RGB) JPEG image files.

31.74.2 Constructor & Destructor Documentation

Fl_JPEG_Image::Fl_JPEG_Image (const char ∗ *filename*)

The constructor loads the JPEG image from the given jpeg filename.

The inherited destructor frees all memory and server resources that are used by the image.

Use Fl_Image::fail() to check if Fl_JPEG_Image failed to load. fail() returns ERR_FILE_ACCESS if the file could not be opened or read, ERR_FORMAT if the JPEG format could not be decoded, and ERR_NO_IMAGE if the image could not be loaded for another reason. If the image has loaded correctly, w(), h(), and d() should return values greater than zero.

Parameters

in	*filename*	a full path and name pointing to a valid jpeg file.

Fl_JPEG_Image::Fl_JPEG_Image (const char ∗ *name*, const unsigned char ∗ *data*)

The constructor loads the JPEG image from memory.

Construct an image from a block of memory inside the application. Fluid offers "binary Data" chunks as a great way to add image data into the C++ source code. name_png can be NULL. If a name is given, the image is added to the list of shared images (see: Fl_Shared_Image) and will be available by that name.

The inherited destructor frees all memory and server resources that are used by the image.

Use Fl_Image::fail() to check if Fl_JPEG_Image failed to load. fail() returns ERR_FILE_ACCESS if the file could not be opened or read, ERR_FORMAT if the JPEG format could not be decoded, and ERR_NO_IMAGE if the image could not be loaded for another reason. If the image has loaded correctly, w(), h(), and d() should return values greater than zero.

Parameters

name	A unique name or NULL
data	A pointer to the memory location of the JPEG image

The documentation for this class was generated from the following files:

- Fl_JPEG_Image.H
- Fl_JPEG_Image.cxx

31.75 Fl_Label Struct Reference

This struct stores all information for a text or mixed graphics label.

`#include <Fl_Widget.H>`

Public Member Functions

- void draw (int, int, int, int, Fl_Align) const

  Draws the label aligned to the given box.
- void measure (int &w, int &h) const

  Measures the size of the label.

Public Attributes

- Fl_Align align_

  alignment of label
- Fl_Color color

  text color
- Fl_Image ∗ deimage

optional image for a deactivated label

- Fl_Font font

  label font used in text

- Fl_Image ∗ image

  optional image for an active label

- Fl_Fontsize size

  size of label font

- uchar type

  type of label.

- const char ∗ value

  label text

31.75.1 Detailed Description

This struct stores all information for a text or mixed graphics label.

Todo There is an aspiration that the Fl_Label type will become a widget by itself. That way we will be avoiding a lot of code duplication by handling labels in a similar fashion to widgets containing text. We also provide an easy interface for very complex labels, containing html or vector graphics. However, this re-factoring is not in place in this release.

31.75.2 Member Function Documentation

void Fl_Label::draw (int *X*, int *Y*, int *W*, int *H*, Fl_Align *align*) const

Draws the label aligned to the given box.

Draws a label with arbitrary alignment in an arbitrary box.

void Fl_Label::measure (int & *W*, int & *H*) const

Measures the size of the label.

Parameters

`in,out`	*W,H*	: this is the requested size for the label text plus image; on return, this will contain the size needed to fit the label

31.75.3 Member Data Documentation

uchar Fl_Label::type

type of label.

See Also

Fl_Labeltype

The documentation for this struct was generated from the following files:

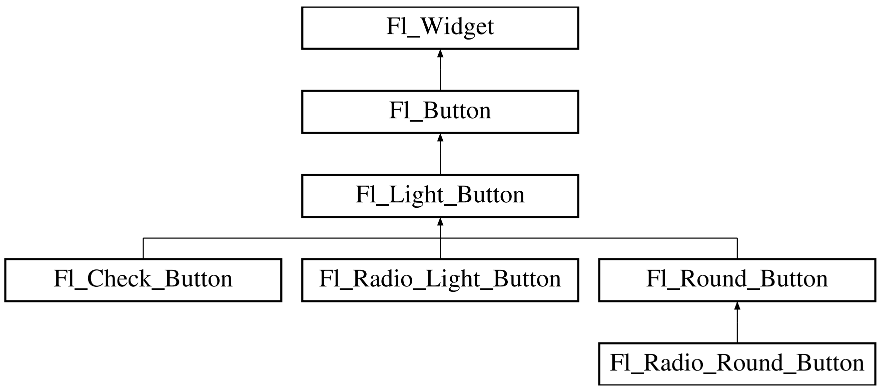
- Fl_Widget.H
- fl_labeltype.cxx

31.76 Fl_Light_Button Class Reference

This subclass displays the "on" state by turning on a light, rather than drawing pushed in.

```
#include <Fl_Light_Button.H>
```

Inheritance diagram for Fl_Light_Button:



Public Member Functions

- Fl_Light_Button (int x, int y, int w, int h, const char ∗l=0)

  Creates a new Fl_Light_Button widget using the given position, size, and label string.
- virtual int handle (int)

  Handles the specified event.

Protected Member Functions

- virtual void draw ()

  Draws the widget.

Additional Inherited Members

31.76.1 Detailed Description

This subclass displays the "on" state by turning on a light, rather than drawing pushed in.

The shape of the "light" is initially set to FL_DOWN_BOX. The color of the light when on is controlled with selection_color(), which defaults to FL_YELLOW.

Buttons generate callbacks when they are clicked by the user. You control exactly when and how by changing the values for type() and when().

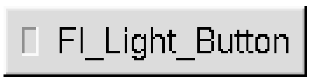


Figure 31.18: Fl_Light_Button

31.76.2 Constructor & Destructor Documentation

Fl_Light_Button::Fl_Light_Button (int *X*, int *Y*, int *W*, int *H*, const char ∗ *l* = *0*)

Creates a new Fl_Light_Button widget using the given position, size, and label string.

The destructor deletes the check button.

31.76.3 Member Function Documentation

void Fl_Light_Button::draw ( ) `[protected]`, `[virtual]`

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call redraw() instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw() method*, e.g. for an embedded scrollbar, you can do it (because draw() is virtual) like this:

```
Fl_Widget *s = &scroll;                // scroll is an embedded Fl_Scrollbar
s->draw();                  // calls Fl_Scrollbar::draw()
```

Reimplemented from Fl_Button.

int Fl_Light_Button::handle (int *event*) `[virtual]`

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited handle() method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters

`in`	*event*	the kind of event received

Return values

0	if the event was not used or understood
1	if the event was used and can be deleted

See Also

Fl_Event

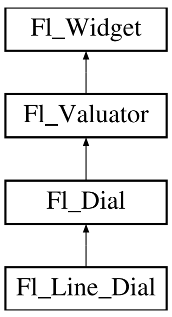
Reimplemented from Fl_Button.

The documentation for this class was generated from the following files:

- Fl_Light_Button.H
- Fl_Light_Button.cxx

31.77 Fl_Line_Dial Class Reference

Inheritance diagram for Fl_Line_Dial:



Public Member Functions

- **Fl_Line_Dial** (int X, int Y, int W, int H, const char ∗L=0)

Additional Inherited Members

The documentation for this class was generated from the following files:

- Fl_Line_Dial.H
- Fl_Dial.cxx

31.78 Fl_Mac_App_Menu Class Reference

Mac OS-specific class allowing to customize and localize the application menu.

Static Public Member Functions

- static void custom_application_menu_items (const Fl_Menu_Item ∗m)

  Adds custom menu item(s) to the application menu of the system menu bar.

Static Public Attributes

- static const char ∗ about = ”About %@”

  Localizable text for the ”About xxx” application menu item.
- static const char ∗ hide = ”Hide %@”

  Localizable text for the ”Hide xxx” application menu item.
- static const char ∗ hide_others = ”Hide Others”

  Localizable text for the ”Hide Others” application menu item.
- static const char ∗ print = ”Print Front Window”

  Localizable text for the ”Print Front Window” application menu item.
- static const char ∗ quit = ”Quit %@”

  Localizable text for the ”Quit xxx” application menu item.
- static const char ∗ services = ”Services”

  Localizable text for the ”Services” application menu item.
- static const char ∗ show = ”Show All”

  Localizable text for the ”Show All” application menu item.

31.78.1 Detailed Description

Mac OS-specific class allowing to customize and localize the application menu.

The public class attributes are used to build the application menu. They can be localized at run time to any UTF-8 text by placing instructions such as this before fl_open_display() gets called:

```
Fl_Mac_App_Menu::print = "Imprimer la fenêtre";
```

See Also

The Apple OS X Interface for another way to localization.

31.78.2 Member Function Documentation

void Fl_Mac_App_Menu::custom_application_menu_items (const Fl_Menu_Item ∗ *m*) `[static]`

Adds custom menu item(s) to the application menu of the system menu bar.

They are positioned after the ”Print Front Window” item, or at its place if it was removed with `Fl_Mac_App_Menu::print = ""`.

Parameters

m	zero-ending array of Fl_Menu_Item 's.

31.78.3 Member Data Documentation

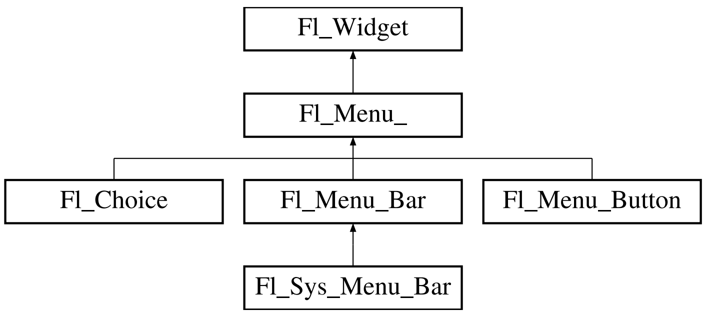
const char ∗ Fl_Mac_App_Menu::print = "Print Front Window" `[static]`

Localizable text for the "Print Front Window" application menu item.
This menu item won't be displayed if Fl_Mac_App_Menu::print is set to an empty string.
The documentation for this class was generated from the following files:

- mac.H
- Fl.cxx
- Fl_Sys_Menu_Bar.mm

31.79 Fl_Menu_ Class Reference

Base class of all widgets that have a menu in FLTK.
`#include <Fl_Menu_.H>`
Inheritance diagram for Fl_Menu_:



Public Member Functions

- int add (const char ∗, int shortcut, Fl_Callback ∗, void ∗=0, int=0)

  Adds a new menu item.
- int add (const char ∗a, const char ∗b, Fl_Callback ∗c, void ∗d=0, int e=0)

  See int Fl_Menu_::add(const char∗ label, int shortcut, Fl_Callback∗, void ∗user_data=0, int flags=0)
- int add (const char ∗)

  This is a Forms (and SGI GL library) compatible add function, it adds many menu items, with '|' separating the menu items, and tab separating the menu item names from an optional shortcut string.
- void clear ()

  Same as menu(NULL), set the array pointer to null, indicating a zero-length menu.
- int clear_submenu (int index)

  Clears the specified submenu pointed to by `index` of all menu items.
- void copy (const Fl_Menu_Item ∗m, void ∗user_data=0)

  Sets the menu array pointer with a copy of m that will be automatically deleted.
- Fl_Boxtype down_box () const

  This box type is used to surround the currently-selected items in the menus.
- void down_box (Fl_Boxtype b)

See Fl_Boxtype Fl_Menu_::down_box() const.

- Fl_Color down_color () const

  For back compatibility, same as selection_color()

- void down_color (unsigned c)

  For back compatibility, same as selection_color()

- int find_index (const char ∗name) const

  Find the menu item index for a given menu `pathname`, such as "Edit/Copy".

- int find_index (const Fl_Menu_Item ∗item) const

  Find the index into the menu array for a given `item`.

- int find_index (Fl_Callback ∗cb) const

  Find the index into the menu array for a given callback `cb`.

- const Fl_Menu_Item ∗ find_item (const char ∗name)

  Find the menu item for a given menu `pathname`, such as "Edit/Copy".

- const Fl_Menu_Item ∗ find_item (Fl_Callback ∗)

  Find the menu item for the given callback `cb`.

- Fl_Menu_ (int, int, int, int, const char ∗=0)

  Creates a new Fl_Menu_ widget using the given position, size, and label string.

- void global ()

  Make the shortcuts for this menu work no matter what window has the focus when you type it.

- int insert (int index, const char ∗, int shortcut, Fl_Callback ∗, void ∗=0, int=0)

  Inserts a new menu item at the specified `index` position.

- int insert (int index, const char ∗a, const char ∗b, Fl_Callback ∗c, void ∗d=0, int e=0)

  See int Fl_Menu_::insert(const char∗ label, int shortcut, Fl_Callback∗, void ∗user_data=0, int flags=0)

- int item_pathname (char ∗name, int namelen, const Fl_Menu_Item ∗finditem=0) const

  Get the menu 'pathname' for the specified menuitem.

- const Fl_Menu_Item ∗ menu () const

  Returns a pointer to the array of Fl_Menu_Items.

- void menu (const Fl_Menu_Item ∗m)

  Sets the menu array pointer directly.

- void mode (int i, int fl)

  Sets the flags of item i.

- int mode (int i) const

  Gets the flags of item i.

- const Fl_Menu_Item ∗ mvalue () const

  Returns a pointer to the last menu item that was picked.

- const Fl_Menu_Item ∗ picked (const Fl_Menu_Item ∗)

  When user picks a menu item, call this.

- void remove (int)

  Deletes item `i` from the menu.

- void replace (int, const char ∗)

  Changes the text of item `i`.

- void setonly (Fl_Menu_Item ∗item)

  Turns the radio item "on" for the menu item and turns "off" adjacent radio items of the same group.

- void shortcut (int i, int s)

  Changes the shortcut of item `i` to `s`.

- int size () const

This returns the number of Fl_Menu_Item structures that make up the menu, correctly counting submenus.

- void **size** (int W, int H)
- const Fl_Menu_Item ∗ test_shortcut ()

  Returns the menu item with the entered shortcut (key value).
- const char ∗ text () const

  Returns the title of the last item chosen.
- const char ∗ text (int i) const

  Returns the title of item i.
- Fl_Color textcolor () const

  Get the current color of menu item labels.
- void textcolor (Fl_Color c)

  Sets the current color of menu item labels.
- Fl_Font textfont () const

  Gets the current font of menu item labels.
- void textfont (Fl_Font c)

  Sets the current font of menu item labels.
- Fl_Fontsize textsize () const

  Gets the font size of menu item labels.
- void textsize (Fl_Fontsize c)

  Sets the font size of menu item labels.
- int value () const

  Returns the index into menu() of the last item chosen by the user.
- int value (const Fl_Menu_Item ∗)

  The value is the index into menu() of the last item chosen by the user.
- int value (int i)

  The value is the index into menu() of the last item chosen by the user.

Protected Member Functions

- int **item_pathname_** (char ∗name, int namelen, const Fl_Menu_Item ∗finditem, const Fl_Menu_Item ∗menu=0) const

Protected Attributes

- uchar **alloc**
- uchar **down_box_**
- Fl_Color **textcolor_**
- Fl_Font **textfont_**
- Fl_Fontsize **textsize_**

Additional Inherited Members

31.79.1 Detailed Description

Base class of all widgets that have a menu in FLTK.

Currently FLTK provides you with Fl_Menu_Button, Fl_Menu_Bar, and Fl_Choice.

The class contains a pointer to an array of structures of type Fl_Menu_Item. The array may either be supplied directly by the user program, or it may be "private": a dynamically allocated array managed by the Fl_Menu_.

When the user clicks a menu item, value() is set to that item and then:

- If the Fl_Menu_Item has a callback set, that callback is invoked with any userdata configured for it. (The Fl_Menu_ widget's callback is NOT invoked.)
- For any Fl_Menu_Items that **don't** have a callback set, the Fl_Menu_ widget's callback is invoked with any userdata configured for it. The callback can determine which item was picked using value(), mvalue(), item_pathname(), etc.

31.79.2 Constructor & Destructor Documentation

Fl_Menu_::Fl_Menu_ (int *X*, int *Y*, int *W*, int *H*, const char ∗ *l* = 0)

Creates a new Fl_Menu_ widget using the given position, size, and label string.
menu() is initialized to null.

31.79.3 Member Function Documentation

int Fl_Menu_::add (const char ∗ *label*, int *shortcut*, Fl_Callback ∗ *callback*, void ∗ *userdata* = 0, int *flags* = 0)

Adds a new menu item.
Parameters

in	*label*	The text label for the menu item.
in	*shortcut*	Optional keyboard shortcut that can be an int or string: (FL_CTRL+'a') or "^a". Default 0 if none.
in	*callback*	Optional callback invoked when user clicks the item. Default 0 if none.
in	*userdata*	Optional user data passed as an argument to the callback. Default 0 if none.
in	*flags*	Optional flags that control the type of menu item; see below. Default is 0 for none.

Returns

The index into the menu() array, where the entry was added.

Description

If the menu array was directly set with menu(x), then copy() is done to make a private array.

Since this method can change the internal menu array, any menu item pointers or indices the application may have cached can become stale, and should be recalculated/refreshed.

A menu item's callback must not add() items to its parent menu during the callback.

Detailed Description of Parameters

label

The menu item's label. This argument is required and must not be NULL.

The characters "&", "/", "\", and "_" are treated as special characters in the label string. The "&" character specifies that the following character is an accelerator and will be underlined. The "\" character is used to escape the next character in the string. Labels starting with the "_" character cause a divider to be placed after that menu item.

A label of the form "File/Quit" will create the submenu "File" with a menu item called "Quit". The "/" character is ignored if it appears as the first character of the label string, e.g. "/File/Quit".

The label string is copied to new memory and can be freed. The other arguments (including the shortcut) are copied into the menu item unchanged.

If an item exists already with that name then it is replaced with this new one. Otherwise this new one is added to the end of the correct menu or submenu. The return value is the offset into the array that the new entry was placed at.

shortcut

The keyboard shortcut for this menu item.

This parameter is optional, and defaults to 0 to indicate no shortcut.

The shortcut can either be a raw integer value (eg. FL_CTRL+'A') or a string (eg. "^c" or "^97").

Raw integer shortcuts can be a combination of keyboard chars (eg. 'A') and optional keyboard modifiers (see Fl::event_state(), e.g. FL_SHIFT, etc). In addition, FL_COMMAND can be used to denote FL_META under Mac OS X and FL_CTRL under other platforms.

String shortcuts can be specified in one of two ways:

```
[#+^]<ascii_value>    e.g. "97", "^97", "+97", "#97"
[#+^]<ascii_char>     e.g. "a", "^a", "+a", "#a"
```

..where <ascii_value> is a decimal value representing an ASCII character (eg. 97 is the ascii code for 'a'), and the optional prefixes enhance the value that follows. Multiple prefixes must appear in the order below.

```
# - Alt
+ - Shift
^ - Control
```

Internally, the text shortcuts are converted to integer values using fl_old_shortcut(const char*).

callback

The callback to invoke when this menu item is selected.

This parameter is optional, and defaults to 0 for no callback.

userdata

The callback's 'user data' that is passed to the callback.

This parameter is optional, and defaults to 0.

flags

These are bit flags to define what kind of menu item this is.

This parameter is optional, and defaults to 0 to define a 'regular' menu item.

These flags can be 'OR'ed together:

```
FL_MENU_INACTIVE     // Deactivate menu item (gray out)
FL_MENU_TOGGLE       // Item is a checkbox toggle (shows checkbox for on/off state)
FL_MENU_VALUE        // The on/off state for checkbox/radio buttons (if set, state is 'on')
FL_MENU_RADIO        // Item is a radio button (one checkbox of many can be on)
FL_MENU_INVISIBLE    // Item will not show up (shortcut will work)
FL_SUBMENU_POINTER   // Indicates user_data() is a pointer to another menu array
FL_SUBMENU            // This item is a submenu to other items
FL_MENU_DIVIDER      // Creates divider line below this item. Also ends a group of radio
       buttons.
```

If FL_SUBMENU is set in an item's flags, then actually two items are added: the first item is the menu item (submenu title), as expected, and the second item is the submenu terminating item with the label and all other members set to 0. If you add submenus with the 'path' technique, then the corresponding submenu terminators (maybe more than one) are added as well.

Todo Raw integer shortcut needs examples. Dependent on responses to `http://fltk.org/newsgroups.php?gfltk.development+v:10086` and results of STR#2344

int Fl_Menu_::add (const char ∗ *str*)

This is a Forms (and SGI GL library) compatible add function, it adds many menu items, with '|' separating the menu items, and tab separating the menu item names from an optional shortcut string.

The passed string is split at any '|' characters and then add(s,0,0,0,0) is done with each section. This is often useful if you are just using the value, and is compatible with Forms and other GL programs. The section strings use the same special characters as described for the long version of add().

No items must be added to a menu during a callback to the same menu.

Parameters

str	string containing multiple menu labels as described above

Returns

the index into the menu() array, where the entry was added

void Fl_Menu_::clear ( )

Same as menu(NULL), set the array pointer to null, indicating a zero-length menu.

Menus must not be cleared during a callback to the same menu.

int Fl_Menu_::clear_submenu (int *index*)

Clears the specified submenu pointed to by `index` of all menu items.

This method is useful for clearing a submenu so that it can be re-populated with new items. Example: a "File/Recent Files/..." submenu that shows the last few files that have been opened.

The specified `index` must point to a submenu.

The submenu is cleared with remove(). If the menu array was directly set with menu(x), then copy() is done to make a private array.

Warning

Since this method can change the internal menu array, any menu item pointers or indices the application may have cached can become stale, and should be recalculated/refreshed.

Example:

```
int index = menubar->find_index("File/Recent");     // get index of "File/Recent" submenu
if ( index != -1 ) menubar->clear_submenu(index);  // clear the submenu
menubar->add("File/Recent/Aaa");
menubar->add("File/Recent/Bbb");
[..]
```

Parameters

index	The index of the submenu to be cleared

Returns

0 on success, -1 if the index is out of range or not a submenu

See Also

remove(int)

void Fl_Menu_::copy (const Fl_Menu_Item ∗ *m*, void ∗ *ud* = 0)

Sets the menu array pointer with a copy of m that will be automatically deleted.

If userdata `ud` is not NULL, then all user data pointers are changed in the menus as well. See void Fl_Menu_::menu(const Fl_Menu_Item∗ m).

Fl_Boxtype Fl_Menu_::down_box ( ) const [inline]

This box type is used to surround the currently-selected items in the menus.

If this is FL_NO_BOX then it acts like FL_THIN_UP_BOX and selection_color() acts like FL_WHITE, for back compatibility.

int Fl_Menu_::find_index (const char ∗ *pathname*) const

Find the menu item index for a given menu `pathname`, such as "Edit/Copy".

This method finds a menu item's index position for the given menu pathname, also traversing submenus, but **not** submenu pointers (FL_SUBMENU_POINTER).

To get the menu item pointer for a pathname, use find_item()

Parameters

`in`	*pathname*	The path and name of the menu item to find

Returns

The index of the matching item, or -1 if not found.

See Also

item_pathname()

int Fl_Menu_::find_index (const Fl_Menu_Item ∗ *item*) const

Find the index into the menu array for a given `item`.

A way to convert a menu item pointer into an index.

Does **not** handle items that are in submenu pointers (FL_SUBMENU_POINTER).

-1 is returned if the item is not in this menu or is part of an FL_SUBMENU_POINTER submenu.

Current implementation is fast and not expensive.

```
// Convert an index-to-item
int index = 12;
const Fl_Menu_Item *item = mymenu->menu() + index;

// Convert an item-to-index
int index = mymenu->find_index(item);
if ( index == -1 ) { ..error.. }
```

Parameters

in	*item*	The item to be found

Returns

The index of the item, or -1 if not found.

See Also

menu()

int Fl_Menu_::find_index (Fl_Callback ∗ *cb*) const

Find the index into the menu array for a given callback cb.

This method finds a menu item's index position, also traversing submenus, but **not** submenu pointers (FL_SUBMENU_POINTER). This is useful if an application uses internationalisation and a menu item can not be found using its label. This search is also much faster.

Parameters

cb	Find the first item with this callback

Returns

The index of the item with the specific callback, or -1 if not found

See Also

find_index(const char∗)

const Fl_Menu_Item ∗ Fl_Menu_::find_item (const char ∗ *pathname*)

Find the menu item for a given menu pathname, such as "Edit/Copy".

This method finds a menu item in the menu array, also traversing submenus, but not submenu pointers (FL_SUBMENU_POINTER).

To get the menu item's index, use find_index(const char∗)

Example:

```
Fl_Menu_Bar *menubar = new Fl_Menu_Bar(..);
menubar->add("File/&Open");
menubar->add("File/&Save");
menubar->add("Edit/&Copy");
// [..]
Fl_Menu_Item *item;
if ( ( item = (Fl_Menu_Item*)menubar->find_item("File/&Open") ) != NULL ) {
    item->labelcolor(FL_RED);
}
if ( ( item = (Fl_Menu_Item*)menubar->find_item("Edit/&Copy") ) != NULL ) {
    item->labelcolor(FL_GREEN);
}
```

Parameters

pathname	The path and name of the menu item

Returns

The item found, or NULL if not found

See Also

find_index(const char∗), find_item(Fl_Callback∗), item_pathname()

const Fl_Menu_Item * Fl_Menu_::find_item (Fl_Callback * *cb*)

Find the menu item for the given callback `cb`.

This method finds a menu item in a menu array, also traversing submenus, but not submenu pointers. This is useful if an application uses internationalisation and a menu item can not be found using its label. This search is also much faster.

Parameters

`in`	*cb*	find the first item with this callback

Returns

The item found, or NULL if not found

See Also

find_item(const char*)

void Fl_Menu_::global ( )

Make the shortcuts for this menu work no matter what window has the focus when you type it.

This is done by using Fl::add_handler(). This Fl_Menu_ widget does not have to be visible (ie the window it is in can be hidden, or it does not have to be put in a window at all).

Currently there can be only one global()menu. Setting a new one will replace the old one. There is no way to remove the global() setting (so don't destroy the widget!)

int Fl_Menu_::insert (int *index*, const char * *label*, int *shortcut*, Fl_Callback * *callback*, void * *userdata* = 0, int *flags* = 0)

Inserts a new menu item at the specified `index` position.

If `index` is -1, the menu item is appended; same behavior as add().

To properly insert a menu item, `label` must be the name of the item (eg. "Quit"), and not a 'menu pathname' (eg. "File/Quit"). If a menu pathname is specified, the value of `index` is *ignored*, the new item's position defined by the pathname.

For more details, see add(). Except for the `index` parameter, add() has more detailed information on parameters and behavior, and is functionally equivalent.

Parameters

`in`	*index*	The menu array's index position where the new item is inserted. If -1, behavior is the same as add().
`in`	*label*	The text label for the menu item. If the label is a menu pathname, `index` is ignored, and the pathname indicates the position of the new item.
`in`	*shortcut*	Optional keyboard shortcut. Can be an int (FL_CTRL+'a') or a string ("^a"). Default is 0.
`in`	*callback*	Optional callback invoked when user clicks the item. Default 0 if none.
`in`	*userdata*	Optional user data passed as an argument to the callback. Default 0 if none.
`in`	*flags*	Optional flags that control the type of menu item; see add() for more info. Default is 0 for none.

Returns

The index into the menu() array, where the entry was added.

See Also

add()

int Fl_Menu_::item_pathname (char ∗ *name*, int *namelen*, const Fl_Menu_Item ∗ *finditem* = 0) const

Get the menu 'pathname' for the specified menuitem.

If finditem==NULL, mvalue() is used (the most recently picked menuitem).

Example:

```
Fl_Menu_Bar *menubar = 0;
void my_menu_callback(Fl_Widget*,void*) {
  char name[80];
  if ( menubar->item_pathname(name, sizeof(name)-1) == 0 ) {   // recently picked item
    if ( strcmp(name, "File/&Open") == 0 ) { .. }              // open invoked
    if ( strcmp(name, "File/&Save") == 0 ) { .. }              // save invoked
    if ( strcmp(name, "Edit/&Copy") == 0 ) { .. }              // copy invoked
  }
}
int main() {
  [..]
  menubar = new Fl_Menu_Bar(..);
  menubar->add("File/&Open",  0, my_menu_callback);
  menubar->add("File/&Save",  0, my_menu_callback);
  menubar->add("Edit/&Copy",  0, my_menu_callback);
  [..]
}
```

Returns

- 0 : OK (name has menuitem's pathname)
- -1 : item not found (name="")
- -2 : 'name' not large enough (name="")

See Also

find_item()

const Fl_Menu_Item∗ Fl_Menu_::menu ( ) const [inline]

Returns a pointer to the array of Fl_Menu_Items.

This will either be the value passed to menu(value) or the private copy.

See Also

size() – returns the size of the Fl_Menu_Item array.

Example: How to walk the array:

```
for ( int t=0; t<menubar->size(); t++ ) {                // walk array of items
    const Fl_Menu_Item &item = menubar->menu()[t];       // get each item
    fprintf(stderr, "item #%d -- label=%s, value=%s type=%s\n",
        t,
        item.label() ? item.label() : "(Null)",          // menu terminators have NULL labels
        (item.flags & FL_MENU_VALUE) ? "set" : "clear",  // value of toggle or radio
       items
        (item.flags & FL_SUBMENU) ? "Submenu" : "Item"); // see if item is a submenu or
       actual item
}
```

void Fl_Menu_::menu (const Fl_Menu_Item ∗ *m*)

Sets the menu array pointer directly.

If the old menu is private it is deleted. NULL is allowed and acts the same as a zero-length menu. If you try to modify the array (with add(), replace(), or remove()) a private copy is automatically done.

void Fl_Menu_::mode (int *i*, int *fl*) `[inline]`

Sets the flags of item i.

For a list of the flags, see Fl_Menu_Item.

int Fl_Menu_::mode (int *i*) const `[inline]`

Gets the flags of item i.

For a list of the flags, see Fl_Menu_Item.

const Fl_Menu_Item∗ Fl_Menu_::mvalue ( ) const `[inline]`

Returns a pointer to the last menu item that was picked.

const Fl_Menu_Item ∗ Fl_Menu_::picked (const Fl_Menu_Item ∗ *v*)

When user picks a menu item, call this.

It will do the callback. Unfortunately this also casts away const for the checkboxes, but this was necessary so non-checkbox menus can really be declared const...

void Fl_Menu_::remove (int *i*)

Deletes item `i` from the menu.

If the menu array was directly set with menu(x) then copy() is done to make a private array.

No items must be removed from a menu during a callback to the same menu.

Parameters

i	index into menu array

void Fl_Menu_::replace (int *i*, const char ∗ *str*)

Changes the text of item `i`.

This is the only way to get slash into an add()'ed menu item. If the menu array was directly set with menu(x) then copy() is done to make a private array.

Parameters

i	index into menu array
str	new label for menu item at index i

void Fl_Menu_::setonly (Fl_Menu_Item ∗ *item*)

Turns the radio item "on" for the menu item and turns "off" adjacent radio items of the same group.

void Fl_Menu_::shortcut (int *i*, int *s*) `[inline]`

Changes the shortcut of item `i` to `s`.

int Fl_Menu_::size ( ) const

This returns the number of Fl_Menu_Item structures that make up the menu, correctly counting submenus.

This includes the "terminator" item at the end. To copy a menu array you need to copy size()∗sizeof(Fl-_Menu_Item) bytes. If the menu is NULL this returns zero (an empty menu will return 1).

const Fl_Menu_Item∗ Fl_Menu_::test_shortcut ( ) `[inline]`

Returns the menu item with the entered shortcut (key value).

This searches the complete menu() for a shortcut that matches the entered key value. It must be called for a FL_KEYBOARD or FL_SHORTCUT event.

If a match is found, the menu's callback will be called.

Returns

matched Fl_Menu_Item or NULL.

const char∗ Fl_Menu_::text ( ) const `[inline]`

Returns the title of the last item chosen.

const char∗ Fl_Menu_::text (int *i*) const `[inline]`

Returns the title of item i.

Fl_Color Fl_Menu_::textcolor ( ) const `[inline]`

Get the current color of menu item labels.

void Fl_Menu_::textcolor (Fl_Color *c*) `[inline]`

Sets the current color of menu item labels.

Fl_Font Fl_Menu_::textfont ( ) const `[inline]`

Gets the current font of menu item labels.

void Fl_Menu_::textfont (Fl_Font *c*) `[inline]`

Sets the current font of menu item labels.

Fl_Fontsize Fl_Menu_::textsize ( ) const `[inline]`

Gets the font size of menu item labels.

void Fl_Menu_::textsize (Fl_Fontsize *c*) `[inline]`

Sets the font size of menu item labels.

int Fl_Menu_::value ( ) const `[inline]`

Returns the index into menu() of the last item chosen by the user.

It is zero initially.

int Fl_Menu_::value (const Fl_Menu_Item ∗ *m*)

The value is the index into menu() of the last item chosen by the user.

It is zero initially. You can set it as an integer, or set it with a pointer to a menu item. The set routines return non-zero if the new value is different than the old one.

int Fl_Menu_::value (int *i*) `[inline]`

The value is the index into menu() of the last item chosen by the user.

It is zero initially. You can set it as an integer, or set it with a pointer to a menu item. The set routines return non-zero if the new value is different than the old one.

The documentation for this class was generated from the following files:

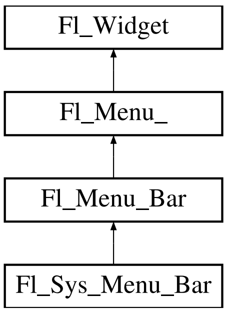
- Fl_Menu_.H
- Fl_Menu_.cxx
- Fl_Menu_add.cxx
- Fl_Menu_global.cxx

31.80 Fl_Menu_Bar Class Reference

This widget provides a standard menubar interface.

`#include <Fl_Menu_Bar.H>`

Inheritance diagram for Fl_Menu_Bar:



Public Member Functions

- Fl_Menu_Bar (int X, int Y, int W, int H, const char ∗l=0)

  Creates a new Fl_Menu_Bar widget using the given position, size, and label string.
- int handle (int)

  Handles the specified event.

Protected Member Functions

- void draw ()

  Draws the widget.

Additional Inherited Members

31.80.1 Detailed Description

This widget provides a standard menubar interface.

Usually you will put this widget along the top edge of your window. The height of the widget should be 30 for the menu titles to draw correctly with the default font.

The items on the bar and the menus they bring up are defined by a single Fl_Menu_Item array. Because a Fl_Menu_Item array defines a hierarchy, the top level menu defines the items in the menubar, while the submenus define the pull-down menus. Sub-sub menus and lower pop up to the right of the submenus.

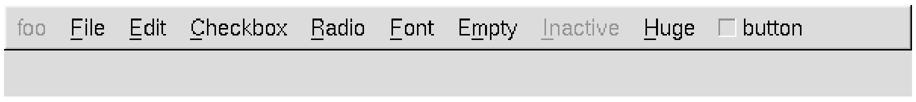


Figure 31.19: menubar

If there is an item in the top menu that is not a title of a submenu, then it acts like a "button" in the menubar. Clicking on it will pick it.

When the user clicks a menu item, value() is set to that item and then:

- The item's callback is done if one has been set; the Fl_Menu_Bar is passed as the Fl_Widget∗ argument, along with any userdata configured for the callback.
- If the item does not have a callback, the Fl_Menu_Bar's callback is done instead, along with any userdata configured for the callback. The callback can determine which item was picked using value(), mvalue(), item_pathname(), etc.

Submenus will also pop up in response to shortcuts indicated by putting a '&' character in the name field of the menu item. If you put a '&' character in a top-level "button" then the shortcut picks it. The '&' character in submenus is ignored until the menu is popped up.

Typing the shortcut() of any of the menu items will cause callbacks exactly the same as when you pick the item with the mouse.

31.80.2 Constructor & Destructor Documentation

Fl_Menu_Bar::Fl_Menu_Bar (int *X*, int *Y*, int *W*, int *H*, const char ∗ *l* = *0*)

Creates a new Fl_Menu_Bar widget using the given position, size, and label string.

The default boxtype is FL_UP_BOX.

The constructor sets menu() to NULL. See Fl_Menu_ for the methods to set or change the menu.

labelsize(), labelfont(), and labelcolor() are used to control how the menubar items are drawn. They are initialized from the Fl_Menu static variables, but you can change them if desired.

label() is ignored unless you change align() to put it outside the menubar.

The destructor removes the Fl_Menu_Bar widget and all of its menu items.

31.80.3 Member Function Documentation

void Fl_Menu_Bar::draw ( ) `[protected]`, `[virtual]`

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call redraw() instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw() method*, e.g. for an embedded scrollbar, you can do it (because draw() is virtual) like this:

```
Fl_Widget *s = &scroll;                // scroll is an embedded Fl_Scrollbar
s->draw();                        // calls Fl_Scrollbar::draw()
```

Implements Fl_Widget.

Reimplemented in Fl_Sys_Menu_Bar.

int Fl_Menu_Bar::handle (int *event*) `[virtual]`

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited handle() method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters

`in`	*event*	the kind of event received

Return values

0	if the event was not used or understood
1	if the event was used and can be deleted

See Also

Fl_Event

Reimplemented from Fl_Widget.

The documentation for this class was generated from the following files:

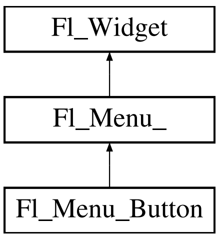
- Fl_Menu_Bar.H
- Fl_Menu_Bar.cxx

31.81 Fl_Menu_Button Class Reference

This is a button that when pushed pops up a menu (or hierarchy of menus) defined by an array of Fl_Menu_Item objects.

`#include <Fl_Menu_Button.H>`

Inheritance diagram for Fl_Menu_Button:



Public Types

- enum popup_buttons {
POPUP1 = 1, POPUP2, POPUP12, POPUP3,
POPUP13, POPUP23, POPUP123 }

  indicate what mouse buttons pop up the menu.

Public Member Functions

- Fl_Menu_Button (int, int, int, int, const char ∗=0)

  Creates a new Fl_Menu_Button widget using the given position, size, and label string.

- int handle (int)

  Handles the specified event.
- const Fl_Menu_Item ∗ popup ()

  Act exactly as though the user clicked the button or typed the shortcut key.

Protected Member Functions

- void draw ()

  Draws the widget.

Additional Inherited Members

31.81.1 Detailed Description

This is a button that when pushed pops up a menu (or hierarchy of menus) defined by an array of Fl_Menu_Item objects.

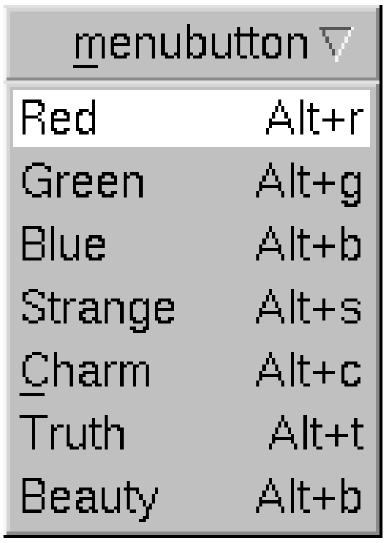


Figure 31.20: menu_button

Normally any mouse button will pop up a menu and it is lined up below the button as shown in the picture. However an Fl_Menu_Button may also control a pop-up menu. This is done by setting the type(). If type() is zero a normal menu button is produced. If it is nonzero then this is a pop-up menu. The bits in type() indicate what mouse buttons pop up the menu (see Fl_Menu_Button::popup_buttons).

The menu will also pop up in response to shortcuts indicated by putting a '&' character in the label().

Typing the shortcut() of any of the menu items will cause callbacks exactly the same as when you pick the item with the mouse. The '&' character in menu item names are only looked at when the menu is popped up, however.

When the user clicks a menu item, value() is set to that item and then:

```
- The item's callback is done if one has been set; the
  Fl_Menu_Button is passed as the Fl_Widget* argument,
  along with any userdata configured for the callback.

- If the item does not have a callback, the Fl_Menu_Button's callback
  is done instead, along with any userdata configured for it.
  The callback can determine which item was picked using
  value(), mvalue(), item_pathname(), etc.
```

31.81.2 Member Enumeration Documentation

enum Fl_Menu_Button::popup_buttons

indicate what mouse buttons pop up the menu.

Values for type() used to indicate what mouse buttons pop up the menu. Fl_Menu_Button::POPUP3 is usually what you want.

Enumerator

POPUP1 pops up with the mouse 1st button.

POPUP2 pops up with the mouse 2nd button.

POPUP12 pops up with the mouse 1st or 2nd buttons.

POPUP3 pops up with the mouse 3rd button.

POPUP13 pops up with the mouse 1st or 3rd buttons.

POPUP23 pops up with the mouse 2nd or 3rd buttons.

POPUP123 pops up with any mouse button.

31.81.3 Constructor & Destructor Documentation

Fl_Menu_Button::Fl_Menu_Button (int *X*, int *Y*, int *W*, int *H*, const char * *l* = *0*)

Creates a new Fl_Menu_Button widget using the given position, size, and label string.

The default boxtype is FL_UP_BOX.

The constructor sets menu() to NULL. See Fl_Menu_ for the methods to set or change the menu.

31.81.4 Member Function Documentation

void Fl_Menu_Button::draw ( ) `[protected]`, `[virtual]`

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call redraw() instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw() method*, e.g. for an embedded scrollbar, you can do it (because draw() is virtual) like this:

```
Fl_Widget *s = &scroll;              // scroll is an embedded Fl_Scrollbar
s->draw();                    // calls Fl_Scrollbar::draw()
```

Implements Fl_Widget.

int Fl_Menu_Button::handle (int *event*) `[virtual]`

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited handle() method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters

in	*event*	the kind of event received

Return values

0	if the event was not used or understood
1	if the event was used and can be deleted

See Also

Fl_Event

Reimplemented from Fl_Widget.

const Fl_Menu_Item ∗ Fl_Menu_Button::popup ( )

Act exactly as though the user clicked the button or typed the shortcut key.

The menu appears, it waits for the user to pick an item, and if they pick one it sets value() and does the callback or sets changed() as described above. The menu item is returned or NULL if the user dismisses the menu.

The documentation for this class was generated from the following files:

- Fl_Menu_Button.H
- Fl_Menu_Button.cxx

31.82 Fl_Menu_Item Struct Reference

The Fl_Menu_Item structure defines a single menu item that is used by the Fl_Menu_ class.

`#include <Fl_Menu_Item.H>`

Public Member Functions

- void activate ()

  Allows a menu item to be picked.
- int active () const

  Gets whether or not the item can be picked.
- int activevisible () const

  Returns non 0 if FL_INACTIVE and FL_INVISIBLE are cleared, 0 otherwise.
- int add (const char ∗, int shortcut, Fl_Callback ∗, void ∗=0, int=0)

  Adds a menu item.
- int add (const char ∗a, const char ∗b, Fl_Callback ∗c, void ∗d=0, int e=0)

  See int add(const char∗, int shortcut, Fl_Callback∗, void∗, int)
- long argument () const

  Gets the user_data() argument that is sent to the callback function.
- void argument (long v)

  Sets the user_data() argument that is sent to the callback function.
- Fl_Callback_p callback () const

  Returns the callback function that is set for the menu item.
- void callback (Fl_Callback ∗c, void ∗p)

  Sets the menu item's callback function and userdata() argument.
- void callback (Fl_Callback ∗c)

  Sets the menu item's callback function.

- void callback (Fl_Callback0 ∗c)

    Sets the menu item's callback function.
- void callback (Fl_Callback1 ∗c, long p=0)

    Sets the menu item's callback function and userdata() argument.
- void check ()

    back compatibility only
- int checkbox () const

    Returns true if a checkbox will be drawn next to this item.
- int checked () const

    back compatibility only
- void clear ()

    Turns the check or radio item "off" for the menu item.
- void deactivate ()

    Prevents a menu item from being picked.
- void do_callback (Fl_Widget ∗o) const

    Calls the Fl_Menu_Item item's callback, and provides the Fl_Widget argument.
- void do_callback (Fl_Widget ∗o, void ∗arg) const

    Calls the Fl_Menu_Item item's callback, and provides the Fl_Widget argument.
- void do_callback (Fl_Widget ∗o, long arg) const

    Calls the Fl_Menu_Item item's callback, and provides the Fl_Widget argument.
- void draw (int x, int y, int w, int h, const Fl_Menu_ ∗, int t=0) const

    Draws the menu item in bounding box x,y,w,h, optionally selects the item.
- const Fl_Menu_Item ∗ find_shortcut (int ∗ip=0, const bool require_alt=false) const

    Search only the top level menu for a shortcut.
- const Fl_Menu_Item ∗ first () const

    Returns the first menu item, same as next(0).
- Fl_Menu_Item ∗ first ()

    Returns the first menu item, same as next(0).
- void hide ()

    Hides an item in the menu.
- void image (Fl_Image ∗a)

    compatibility api for FLUID, same as a->label(this)
- void image (Fl_Image &a)

    compatibility api for FLUID, same as a.label(this)
- int insert (int, const char ∗, int, Fl_Callback ∗, void ∗=0, int=0)

    Inserts an item at position `index`.
- const char ∗ label () const

    Returns the title of the item.
- void label (const char ∗a)

    See const char∗ Fl_Menu_Item::label() const.
- void label (Fl_Labeltype a, const char ∗b)

    See const char∗ Fl_Menu_Item::label() const.
- Fl_Color labelcolor () const

    Gets the menu item's label color.
- void labelcolor (Fl_Color a)

    Sets the menu item's label color.

- Fl_Font labelfont () const

  Gets the menu item's label font.
- void labelfont (Fl_Font a)

  Sets the menu item's label font.
- Fl_Fontsize labelsize () const

  Gets the label font pixel size/height.
- void labelsize (Fl_Fontsize a)

  Sets the label font pixel size/height.
- Fl_Labeltype labeltype () const

  Returns the menu item's labeltype.
- void labeltype (Fl_Labeltype a)

  Sets the menu item's labeltype.
- int measure (int ∗h, const Fl_Menu_ ∗) const

  Measures width of label, including effect of & characters.
- const Fl_Menu_Item ∗ next (int=1) const

  Advance a pointer by n items through a menu array, skipping the contents of submenus and invisible items.
- Fl_Menu_Item ∗ next (int i=1)

  Advances a pointer by n items through a menu array, skipping the contents of submenus and invisible items.
- const Fl_Menu_Item ∗ popup (int X, int Y, const char ∗title=0, const Fl_Menu_Item ∗picked=0, const Fl_Menu_ ∗=0) const

  This method is called by widgets that want to display menus.
- const Fl_Menu_Item ∗ pulldown (int X, int Y, int W, int H, const Fl_Menu_Item ∗picked=0, const Fl_Menu_ ∗=0, const Fl_Menu_Item ∗title=0, int menubar=0) const

  Pulldown() is similar to popup(), but a rectangle is provided to position the menu.
- int radio () const

  Returns true if this item is a radio item.
- void set ()

  Turns the check or radio item "on" for the menu item.
- void setonly ()

  Turns the radio item "on" for the menu item and turns "off" adjacent radio items set.
- int shortcut () const

  Gets what key combination shortcut will trigger the menu item.
- void shortcut (int s)

  Sets exactly what key combination will trigger the menu item.
- void show ()

  Makes an item visible in the menu.
- int size () const

  Size of the menu starting from this menu item.
- int submenu () const

  Returns true if either FL_SUBMENU or FL_SUBMENU_POINTER is on in the flags.
- const Fl_Menu_Item ∗ test_shortcut () const

  This is designed to be called by a widgets handle() method in response to a FL_SHORTCUT event.
- void uncheck ()

  back compatibility only
- void ∗ user_data () const

  Gets the user_data() argument that is sent to the callback function.
- void user_data (void ∗v)

Sets the user_data() argument that is sent to the callback function.

- int value () const

  Returns the current value of the check or radio item.

- int visible () const

  Gets the visibility of an item.

Public Attributes

- Fl_Callback ∗ callback_

  menu item callback

- int flags

  menu item flags like FL_MENU_TOGGLE, FL_MENU_RADIO

- Fl_Color labelcolor_

  menu item text color

- Fl_Font labelfont_

  which font for this menu item text

- Fl_Fontsize labelsize_

  size of menu item text

- uchar labeltype_

  how the menu item text looks like

- int shortcut_

  menu item shortcut

- const char ∗ text

  menu item text, returned by label()

- void ∗ user_data_

  menu item user_data for the menu's callback

31.82.1 Detailed Description

The Fl_Menu_Item structure defines a single menu item that is used by the Fl_Menu_ class.

```
struct Fl_Menu_Item {
 const char*   text;     // label()
 ulong         shortcut_;
 Fl_Callback*  callback_;
 void*         user_data_;
 int           flags;
 uchar         labeltype_;
 uchar         labelfont_;
 uchar         labelsize_;
 uchar         labelcolor_;
};

enum { // values for flags:
 FL_MENU_INACTIVE   = 1,      // Deactivate menu item (gray out)
 FL_MENU_TOGGLE     = 2,      // Item is a checkbox toggle (shows checkbox for on/off state)
 FL_MENU_VALUE      = 4,      // The on/off state for checkbox/radio buttons (if set, state is
       'on')
 FL_MENU_RADIO      = 8,      // Item is a radio button (one checkbox of many can be on)
 FL_MENU_INVISIBLE  = 0x10,   // Item will not show up (shortcut will work)
 FL_SUBMENU_POINTER = 0x20,   // Indicates user_data() is a pointer to another menu array
 FL_SUBMENU          = 0x40,   // This item is a submenu to other items
 FL_MENU_DIVIDER    = 0x80,   // Creates divider line below this item. Also ends a group of
       radio buttons.
 FL_MENU_HORIZONTAL = 0x100   // ??? -- reserved
};
```

Typically menu items are statically defined; for example:

```
Fl_Menu_Item popup[] = {
 {"&alpha",    FL_ALT+'a', the_cb, (void*)1},
 {"&beta",     FL_ALT+'b', the_cb, (void*)2},
 {"gamma",     FL_ALT+'c', the_cb, (void*)3, FL_MENU_DIVIDER},
 {"&strange",  0,          strange_cb},
 {"&charm",    0,          charm_cb},
 {"&truth",    0,          truth_cb},
 {"b&eauty",   0,          beauty_cb},
 {"sub&menu",  0,          0, 0, FL_SUBMENU},
 {"one"},
 {"two"},
 {"three"},
 {0},
 {"inactive", FL_ALT+'i', 0, 0, FL_MENU_INACTIVE|
      FL_MENU_DIVIDER},
 {"invisible",FL_ALT+'i', 0, 0, FL_MENU_INVISIBLE},
 {"check",    FL_ALT+'i', 0, 0, FL_MENU_TOGGLE|FL_MENU_VALUE},
 {"box",      FL_ALT+'i', 0, 0, FL_MENU_TOGGLE},
 {0}};
```

produces:

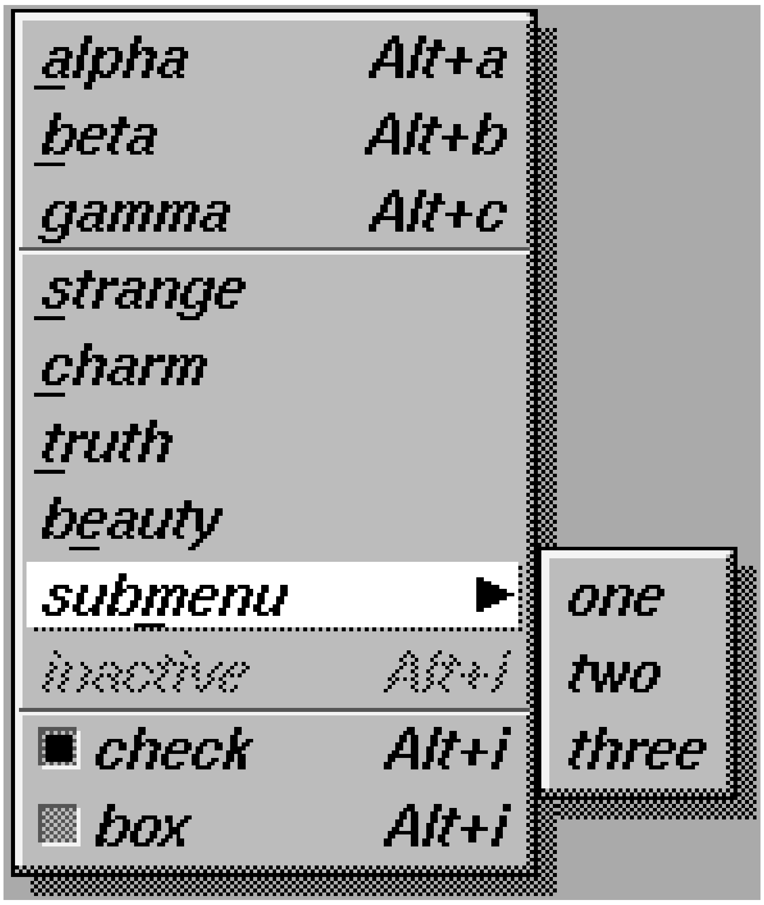


Figure 31.21: menu

A submenu title is identified by the bit FL_SUBMENU in the flags field, and ends with a label() that is NULL. You can nest menus to any depth. A pointer to the first item in the submenu can be treated as an Fl_Menu array itself. It is also possible to make separate submenu arrays with FL_SUBMENU_POINTER flags.

You should use the method functions to access structure members and not access them directly to avoid compatibility problems with future releases of FLTK.

31.82.2 Member Function Documentation

void Fl_Menu_Item::activate ( ) `[inline]`

Allows a menu item to be picked.

int Fl_Menu_Item::active ( ) const `[inline]`

Gets whether or not the item can be picked.

int Fl_Menu_Item::activevisible ( ) const `[inline]`

Returns non 0 if FL_INACTIVE and FL_INVISIBLE are cleared, 0 otherwise.

int Fl_Menu_Item::add (const char ∗ *mytext*, int *sc*, Fl_Callback ∗ *cb*, void ∗ *data* = *0*, int *myflags* = *0*)

Adds a menu item.

The text is split at '/' characters to automatically produce submenus (actually a totally unnecessary feature as you can now add submenu titles directly by setting FL_SUBMENU in the flags).

Returns

the index into the menu() array, where the entry was added

See Also

Fl_Menu_Item::insert(int, const char∗, int, Fl_Callback∗, void∗, int)

long Fl_Menu_Item::argument ( ) const `[inline]`

Gets the user_data() argument that is sent to the callback function.

For convenience you can also define the callback as taking a long argument. This method casts the stored userdata() argument to long and returns it as a *long* value.

void Fl_Menu_Item::argument (long *v*) `[inline]`

Sets the user_data() argument that is sent to the callback function.

For convenience you can also define the callback as taking a long argument. This method casts the given argument `v` to void∗ and stores it in the menu item's userdata() member. This may not be portable to some machines.

Fl_Callback_p Fl_Menu_Item::callback ( ) const `[inline]`

Returns the callback function that is set for the menu item.

Each item has space for a callback function and an argument for that function. Due to back compatibility, the Fl_Menu_Item itself is not passed to the callback, instead you have to get it by calling ((Fl_Menu_∗)w)->mvalue() where w is the widget argument.

void Fl_Menu_Item::callback (Fl_Callback ∗ *c*, void ∗ *p*) `[inline]`

Sets the menu item's callback function and userdata() argument.

See Also

Fl_Callback_p Fl_MenuItem::callback() const

void Fl_Menu_Item::callback (Fl_Callback ∗ *c*) `[inline]`

Sets the menu item's callback function.
This method does not set the userdata() argument.

See Also

Fl_Callback_p Fl_MenuItem::callback() const

void Fl_Menu_Item::callback (Fl_Callback0 ∗ *c*) `[inline]`

Sets the menu item's callback function.
This method does not set the userdata() argument.

See Also

Fl_Callback_p Fl_MenuItem::callback() const

void Fl_Menu_Item::callback (Fl_Callback1 ∗ *c*, long *p* = *0*) `[inline]`

Sets the menu item's callback function and userdata() argument.
This method does not set the userdata() argument. The argument `is` cast to void∗ and stored as the userdata() for the menu item's callback function.

See Also

Fl_Callback_p Fl_MenuItem::callback() const

void Fl_Menu_Item::check ( ) `[inline]`

back compatibility only

Deprecated .

int Fl_Menu_Item::checkbox ( ) const `[inline]`

Returns true if a checkbox will be drawn next to this item.
This is true if FL_MENU_TOGGLE or FL_MENU_RADIO is set in the flags.

int Fl_Menu_Item::checked ( ) const `[inline]`

back compatibility only

Deprecated .

void Fl_Menu_Item::clear ( ) `[inline]`

Turns the check or radio item "off" for the menu item.

void Fl_Menu_Item::deactivate ( ) `[inline]`

Prevents a menu item from being picked.
Note that this will also cause the menu item to appear grayed-out.

void Fl_Menu_Item::do_callback (Fl_Widget ∗ *o*) const `[inline]`

Calls the Fl_Menu_Item item's callback, and provides the Fl_Widget argument.

The callback is called with the stored user_data() as its second argument. You must first check that callback() is non-zero before calling this.

void Fl_Menu_Item::do_callback (Fl_Widget ∗ *o*, void ∗ *arg*) const `[inline]`

Calls the Fl_Menu_Item item's callback, and provides the Fl_Widget argument.

This call overrides the callback's second argument with the given value `arg`. You must first check that callback() is non-zero before calling this.

void Fl_Menu_Item::do_callback (Fl_Widget ∗ *o*, long *arg*) const `[inline]`

Calls the Fl_Menu_Item item's callback, and provides the Fl_Widget argument.

This call overrides the callback's second argument with the given value `arg`. long `arg` is cast to void∗ when calling the callback. You must first check that callback() is non-zero before calling this.

void Fl_Menu_Item::draw (int *x*, int *y*, int *w*, int *h*, const Fl_Menu_ ∗ *m*, int *selected* = `0`) const

Draws the menu item in bounding box x,y,w,h, optionally selects the item.

const Fl_Menu_Item ∗ Fl_Menu_Item::find_shortcut (int ∗ *ip* = `0`, const bool *require_alt* = `false`) const

Search only the top level menu for a shortcut.

Either &x in the label or the shortcut fields are used.

This tests the current event, which must be an FL_KEYBOARD or FL_SHORTCUT, against a shortcut value.

Parameters

ip	returns the index of the item, if `ip` is not NULL.
require_alt	if true: match only if Alt key is pressed.

Returns

found Fl_Menu_Item or NULL

const Fl_Menu_Item∗ Fl_Menu_Item::first ( ) const `[inline]`

Returns the first menu item, same as next(0).

Fl_Menu_Item∗ Fl_Menu_Item::first ( ) `[inline]`

Returns the first menu item, same as next(0).

void Fl_Menu_Item::hide ( ) `[inline]`

Hides an item in the menu.

int Fl_Menu_Item::insert (int *index*, const char ∗ *mytext*, int *sc*, Fl_Callback ∗ *cb*, void ∗ *data* = `0`, int *myflags* = `0`)

Inserts an item at position `index`.

If `index` is -1, the item is added the same way as Fl_Menu_Item::add().

If 'mytext' contains any un-escaped front slashes (/), it's assumed a menu pathname is being specified, and the value of `index` will be ignored.

In all other aspects, the behavior of insert() is the same as add().

Parameters

in	*index*	insert new items here
in	*mytext*	new label string, details see above
in	*sc*	keyboard shortcut for new item
in	*cb*	callback function for new item
in	*data*	user data for new item
in	*myflags*	menu flags as described in FL_Menu_Item

Returns

the index into the menu() array, where the entry was added

const char* Fl_Menu_Item::label ( ) const `[inline]`

Returns the title of the item.

A NULL here indicates the end of the menu (or of a submenu). A '&' in the item will print an underscore under the next letter, and if the menu is popped up that letter will be a "shortcut" to pick that item. To get a real '&' put two in a row.

Fl_Color Fl_Menu_Item::labelcolor ( ) const `[inline]`

Gets the menu item's label color.

This color is passed to the labeltype routine, and is typically the color of the label text. This defaults to FL_BLACK. If this color is not black fltk will **not** use overlay bitplanes to draw the menu - this is so that images put in the menu draw correctly.

void Fl_Menu_Item::labelcolor (Fl_Color *a*) `[inline]`

Sets the menu item's label color.

See Also

Fl_Color Fl_Menu_Item::labelcolor() const

Fl_Font Fl_Menu_Item::labelfont ( ) const `[inline]`

Gets the menu item's label font.

Fonts are identified by small 8-bit indexes into a table. See the enumeration list for predefined fonts. The default value is a Helvetica font. The function Fl::set_font() can define new fonts.

void Fl_Menu_Item::labelfont (Fl_Font *a*) `[inline]`

Sets the menu item's label font.

Fonts are identified by small 8-bit indexes into a table. See the enumeration list for predefined fonts. The default value is a Helvetica font. The function Fl::set_font() can define new fonts.

Fl_Fontsize Fl_Menu_Item::labelsize ( ) const `[inline]`

Gets the label font pixel size/height.

void Fl_Menu_Item::labelsize (Fl_Fontsize *a*) `[inline]`

Sets the label font pixel size/height.

Fl_Labeltype Fl_Menu_Item::labeltype ( ) const `[inline]`

Returns the menu item's labeltype.

A labeltype identifies a routine that draws the label of the widget. This can be used for special effects such as emboss, or to use the label() pointer as another form of data such as a bitmap. The value FL_NORMAL_LABEL prints the label as text.

void Fl_Menu_Item::labeltype (Fl_Labeltype *a*) `[inline]`

Sets the menu item's labeltype.

A labeltype identifies a routine that draws the label of the widget. This can be used for special effects such as emboss, or to use the label() pointer as another form of data such as a bitmap. The value FL_NORMAL_LABEL prints the label as text.

int Fl_Menu_Item::measure (int ∗ *hp*, const Fl_Menu_ ∗ *m*) const

Measures width of label, including effect of & characters.

Optionally, can get height if hp is not NULL.

const Fl_Menu_Item ∗ Fl_Menu_Item::next (int *n* = *1*) const

Advance a pointer by n items through a menu array, skipping the contents of submenus and invisible items.

There are two calls so that you can advance through const and non-const data.

Fl_Menu_Item∗ Fl_Menu_Item::next (int *i* = *1*) `[inline]`

Advances a pointer by n items through a menu array, skipping the contents of submenus and invisible items.

There are two calls so that you can advance through const and non-const data.

const Fl_Menu_Item ∗ Fl_Menu_Item::popup (int *X*, int *Y*, const char ∗ *title* = *0*, const Fl_Menu_Item ∗ *picked* = *0*, const Fl_Menu_ ∗ *button* = *0*) const

This method is called by widgets that want to display menus.

The menu stays up until the user picks an item or dismisses it. The selected item (or NULL if none) is returned. *This does not do the callbacks or change the state of check or radio items.*

X,Y is the position of the mouse cursor, relative to the window that got the most recent event (usually you can pass Fl::event_x() and Fl::event_y() unchanged here).

`title` is a character string title for the menu. If non-zero a small box appears above the menu with the title in it.

The menu is positioned so the cursor is centered over the item picked. This will work even if `picked` is in a submenu. If `picked` is zero or not in the menu item table the menu is positioned with the cursor in the top-left corner.

`button` is a pointer to an Fl_Menu_ from which the color and boxtypes for the menu are pulled. If NULL then defaults are used.

const Fl_Menu_Item ∗ Fl_Menu_Item::pulldown (int *X*, int *Y*, int *W*, int *H*, const Fl_Menu_Item ∗ *initial_item* = *0*, const Fl_Menu_ ∗ *pbutton* = *0*, const Fl_Menu_Item ∗ *t* = *0*, int *menubar* = *0*) const

Pulldown() is similar to popup(), but a rectangle is provided to position the menu.

The menu is made at least W wide, and the picked item is centered over the rectangle (like Fl_Choice uses). If picked is zero or not found, the menu is aligned just below the rectangle (like a pulldown menu).

The title and menubar arguments are used internally by the Fl_Menu_Bar widget.

int Fl_Menu_Item::radio ( ) const [inline]

Returns true if this item is a radio item.

When a radio button is selected all "adjacent" radio buttons are turned off. A set of radio items is delimited by an item that has radio() false, or by an item with FL_MENU_DIVIDER turned on.

void Fl_Menu_Item::set ( ) [inline]

Turns the check or radio item "on" for the menu item.

Note that this does not turn off any adjacent radio items like set_only() does.

void Fl_Menu_Item::setonly ( )

Turns the radio item "on" for the menu item and turns "off" adjacent radio items set.

Deprecated This method is dangerous if radio items are first in the menu. Use Fl_Menu_::setonly(Fl_Menu_Item*) instead.

int Fl_Menu_Item::shortcut ( ) const [inline]

Gets what key combination shortcut will trigger the menu item.

void Fl_Menu_Item::shortcut (int *s*) [inline]

Sets exactly what key combination will trigger the menu item.

The value is a logical 'or' of a key and a set of shift flags, for instance FL_ALT+'a' or FL_ALT+FL_F+10 or just 'a'. A value of zero disables the shortcut.

The key can be any value returned by Fl::event_key(), but will usually be an ASCII letter. Use a lowercase letter unless you require the shift key to be held down.

The shift flags can be any set of values accepted by Fl::event_state(). If the bit is on that shift key must be pushed. Meta, Alt, Ctrl, and Shift must be off if they are not in the shift flags (zero for the other bits indicates a "don't care" setting).

void Fl_Menu_Item::show ( ) [inline]

Makes an item visible in the menu.

int Fl_Menu_Item::size ( ) const

Size of the menu starting from this menu item.

This method counts all menu items starting with `this` menu item, including all menu items in the same (sub)menu level, all nested submenus, **and** the terminating empty (0) menu item.

It does **not** count menu items referred to by FL_SUBMENU_POINTER menu items (except the single menu item with FL_SUBMENU_POINTER).

All menu items counted are consecutive in memory (one array).

Example:

```
schemechoice = new Fl_Choice(X+125,Y,140,25,"FLTK Scheme");
schemechoice->add("none");
schemechoice->add("plastic");
schemechoice->add("gtk+");
schemechoice->add("gleam");
printf("schemechoice->menu()->size() = %d\n", schemechoice->menu()->size());
```

Output:

schemechoice->menu()->size() = 5

int Fl_Menu_Item::submenu ( ) const `[inline]`

Returns true if either FL_SUBMENU or FL_SUBMENU_POINTER is on in the flags.

FL_SUBMENU indicates an embedded submenu that goes from the next item through the next one with a NULL label(). FL_SUBMENU_POINTER indicates that user_data() is a pointer to another menu array.

const Fl_Menu_Item * Fl_Menu_Item::test_shortcut ( ) const

This is designed to be called by a widgets handle() method in response to a FL_SHORTCUT event.

If the current event matches one of the items shortcut, that item is returned. If the keystroke does not match any shortcuts then NULL is returned. This only matches the shortcut() fields, not the letters in the title preceeded by '

void Fl_Menu_Item::uncheck ( ) `[inline]`

back compatibility only

Deprecated .

int Fl_Menu_Item::value ( ) const `[inline]`

Returns the current value of the check or radio item.

This is zero (0) if the menu item is not checked and non-zero otherwise. You should not rely on a particular value, only zero or non-zero.

Note

> The returned value for a checked menu item as of FLTK 1.3.2 is FL_MENU_VALUE (4), but may be 1 in a future version.

int Fl_Menu_Item::visible ( ) const `[inline]`

Gets the visibility of an item.

The documentation for this struct was generated from the following files:

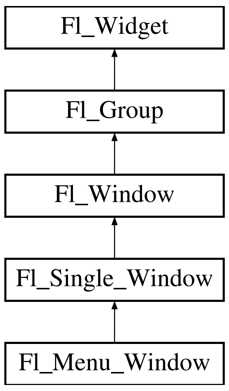
- Fl_Menu_Item.H
- Fl_Menu.cxx
- Fl_Menu_.cxx
- Fl_Menu_add.cxx

31.83 Fl_Menu_Window Class Reference

The Fl_Menu_Window widget is a window type used for menus.

`#include <Fl_Menu_Window.H>`

Inheritance diagram for Fl_Menu_Window:



Public Member Functions

- void clear_overlay ()

  Tells FLTK to use normal drawing planes instead of overlay planes.
- void erase ()

  Erases the window, does nothing if HAVE_OVERLAY is not defined config.h.
- Fl_Menu_Window (int W, int H, const char ∗l=0)

  Creates a new Fl_Menu_Window widget using the given size, and label string.
- Fl_Menu_Window (int X, int Y, int W, int H, const char ∗l=0)

  Creates a new Fl_Menu_Window widget using the given position, size, and label string.
- void flush ()

  Forces the window to be drawn, this window is also made current and calls draw().
- void hide ()

  Removes the window from the screen.
- unsigned int overlay ()

  Tells if hardware overlay mode is set.
- void set_overlay ()

  Tells FLTK to use hardware overlay planes if they are available.
- void show ()

  Puts the window on the screen.
- ∼Fl_Menu_Window ()

  Destroys the window and all of its children.

Additional Inherited Members

31.83.1 Detailed Description

The Fl_Menu_Window widget is a window type used for menus.

By default the window is drawn in the hardware overlay planes if they are available so that the menu don't force the rest of the window to redraw.

31.83.2 Constructor & Destructor Documentation

Fl_Menu_Window::∼Fl_Menu_Window ( )

Destroys the window and all of its children.

Fl_Menu_Window::Fl_Menu_Window (int *W*, int *H*, const char ∗ *l* = `0`)

Creates a new Fl_Menu_Window widget using the given size, and label string.

Fl_Menu_Window::Fl_Menu_Window (int *X*, int *Y*, int *W*, int *H*, const char ∗ *l* = `0`)

Creates a new Fl_Menu_Window widget using the given position, size, and label string.

31.83.3 Member Function Documentation

void Fl_Menu_Window::clear_overlay ( ) `[inline]`

Tells FLTK to use normal drawing planes instead of overlay planes.
This is usually necessary if your menu contains multi-color pixmaps.

void Fl_Menu_Window::flush ( ) `[virtual]`

Forces the window to be drawn, this window is also made current and calls draw().
Reimplemented from Fl_Window.

void Fl_Menu_Window::hide ( ) `[virtual]`

Removes the window from the screen.
If the window is already hidden or has not been shown then this does nothing and is harmless.
Reimplemented from Fl_Window.

void Fl_Menu_Window::set_overlay ( ) `[inline]`

Tells FLTK to use hardware overlay planes if they are available.

void Fl_Menu_Window::show ( ) `[virtual]`

Puts the window on the screen.
Usually (on X) this has the side effect of opening the display.
If the window is already shown then it is restored and raised to the top. This is really convenient because your program can call show() at any time, even if the window is already up. It also means that show() serves the purpose of raise() in other toolkits.
Fl_Window::show(int argc, char ∗∗argv) is used for top-level windows and allows standard arguments to be parsed from the command-line.

Note

For some obscure reasons Fl_Window::show() resets the current group by calling Fl_Group::current(0). The comments in the code say "get rid of very common user bug: forgot end()". Although this is true it may have unwanted side effects if you show() an unrelated window (maybe for an error message or warning) while building a window or any other group widget.

Todo Check if we can remove resetting the current group in a later FLTK version (after 1.3.x). This may break "already broken" programs though if they rely on this "feature".

See Also

Fl_Window::show(int argc, char ∗∗argv)

Reimplemented from Fl_Window.
The documentation for this class was generated from the following files:

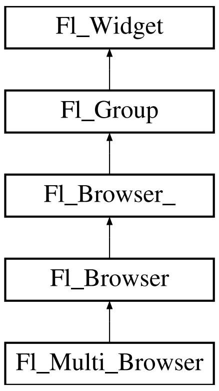
- Fl_Menu_Window.H
- Fl_Menu_Window.cxx

31.84 Fl_Multi_Browser Class Reference

The Fl_Multi_Browser class is a subclass of Fl_Browser which lets the user select any set of the lines.

`#include <Fl_Multi_Browser.H>`

Inheritance diagram for Fl_Multi_Browser:



Public Member Functions

- Fl_Multi_Browser (int X, int Y, int W, int H, const char ∗L=0)

  Creates a new Fl_Multi_Browser widget using the given position, size, and label string.

Additional Inherited Members

31.84.1 Detailed Description

The Fl_Multi_Browser class is a subclass of Fl_Browser which lets the user select any set of the lines.

The user interface is Macintosh style: clicking an item turns off all the others and selects that one, dragging selects all the items the mouse moves over, and ctrl + click (Cmd+click on the Mac OS platform) toggles the items. Shift + click extends the selection until the clicked item. This is different from how forms did it. Normally the callback is done when the user releases the mouse, but you can change this with when().

See Fl_Browser for methods to add and remove lines from the browser.

31.84.2 Constructor & Destructor Documentation

Fl_Multi_Browser::Fl_Multi_Browser (int *X*, int *Y*, int *W*, int *H*, const char ∗ *L* = `0`)

Creates a new Fl_Multi_Browser widget using the given position, size, and label string.

The default boxtype is FL_DOWN_BOX. The constructor specializes Fl_Browser() by setting the type to FL_MULTI_BROWSER. The destructor destroys the widget and frees all memory that has been allocated.

The documentation for this class was generated from the following files:

- Fl_Multi_Browser.H
- Fl_Browser.cxx

31.85 Fl_Multi_Label Struct Reference

This struct allows multiple labels to be added to objects that might normally have only one label.

`#include <Fl_Multi_Label.H>`

Public Member Functions

- void label (Fl_Widget ∗)

  This method is used to associate a Fl_Multi_Label with a Fl_Widget.
- void label (Fl_Menu_Item ∗)

  This method is used to associate a Fl_Multi_Label with a Fl_Menu_Item.

Public Attributes

- const char ∗ labela

  Holds the "leftmost" of the two elements in the composite label.
- const char ∗ labelb

  Holds the "rightmost" of the two elements in the composite label.
- uchar typea

  Holds the "type" of labela.
- uchar typeb

  Holds the "type" of labelb.

31.85.1 Detailed Description

This struct allows multiple labels to be added to objects that might normally have only one label.

This struct allows a mixed text and/or graphics label to be applied to an object that would normally only have a single (usually text only) label.

Most regular FLTK widgets now support the ability to associate both images and text with a label but some special cases, notably the non-widget Fl_Menu_Item objects, do not. Fl_Multi_Label may be used to create menu items that have an icon and text, which would not normally be possible for an Fl_Menu_Item. For example, Fl_Multi_Label is used in the New->Code submenu in fluid, and others.

Each Fl_Multi_Label holds two elements, labela and labelb; each may hold either a text label (const char∗) or an image (Fl_Image∗). When displayed, labela is drawn first and labelb is drawn immediately to its right.

More complex labels might be constructed by setting labelb as another Fl_Multi_Label and thus chaining up a series of label elements.

When assigning a label element to one of labela or labelb, they should be explicitly cast to (const char∗) if they are not of that type already.

See Also

Fl_Label and Fl_Labeltype

31.85.2 Member Function Documentation

void Fl_Multi_Label::label (Fl_Widget ∗ *o*)

This method is used to associate a Fl_Multi_Label with a Fl_Widget.

void Fl_Multi_Label::label (Fl_Menu_Item ∗ *o*)

This method is used to associate a Fl_Multi_Label with a Fl_Menu_Item.

31.85.3 Member Data Documentation

const char∗ Fl_Multi_Label::labela

Holds the "leftmost" of the two elements in the composite label.

Typically this would be assigned either a text string (const char∗), a (Fl_Image∗) or a (Fl_Multi_Label∗).

const char∗ Fl_Multi_Label::labelb

Holds the "rightmost" of the two elements in the composite label.

Typically this would be assigned either a text string (const char∗), a (Fl_Image∗) or a (Fl_Multi_Label∗).

uchar Fl_Multi_Label::typea

Holds the "type" of labela.

Typically this is set to FL_NORMAL_LABEL for a text label, _FL_IMAGE_LABEL for an image (based on Fl_image) or _FL_MULTI_LABEL if "chaining" multiple Fl_Multi_Label elements together.

uchar Fl_Multi_Label::typeb

Holds the "type" of labelb.

Typically this is set to FL_NORMAL_LABEL for a text label, _FL_IMAGE_LABEL for an image (based on Fl_image) or _FL_MULTI_LABEL if "chaining" multiple Fl_Multi_Label elements together.

The documentation for this struct was generated from the following files:

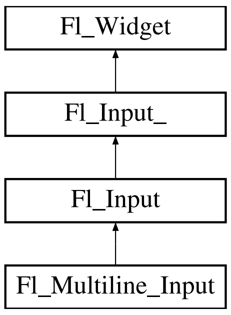
- Fl_Multi_Label.H
- Fl_Multi_Label.cxx

31.86 Fl_Multiline_Input Class Reference

This input field displays '\n' characters as new lines rather than ^J, and accepts the Return, Tab, and up and down arrow keys.

`#include <Fl_Multiline_Input.H>`

Inheritance diagram for Fl_Multiline_Input:



Public Member Functions

- Fl_Multiline_Input (int X, int Y, int W, int H, const char ∗l=0)

  Creates a new Fl_Multiline_Input widget using the given position, size, and label string.

Additional Inherited Members

31.86.1 Detailed Description

This input field displays '\n' characters as new lines rather than ^J, and accepts the Return, Tab, and up and down arrow keys.

This is for editing multiline text.

This is far from the nirvana of text editors, and is probably only good for small bits of text, 10 lines at most. Note that this widget does not support scrollbars or per-character color control.

If you are presenting large amounts of text and need scrollbars or full color control of characters, you probably want Fl_Text_Editor instead.

In FLTK 1.3.x, the default behavior of the 'Tab' key was changed to support consistent focus navigation. To get the older FLTK 1.1.x behavior, set Fl_Input_::tab_nav() to 0. Newer programs should consider using Fl_Text_Editor.

31.86.2 Constructor & Destructor Documentation

Fl_Multiline_Input::Fl_Multiline_Input (int *X*, int *Y*, int *W*, int *H*, const char ∗ *l* = *0*)

Creates a new Fl_Multiline_Input widget using the given position, size, and label string.

The default boxtype is FL_DOWN_BOX.

Inherited destructor destroys the widget and any value associated with it.

The documentation for this class was generated from the following files:

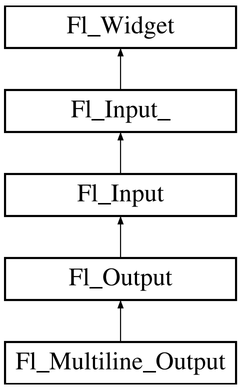
- Fl_Multiline_Input.H
- Fl_Input.cxx

31.87 Fl_Multiline_Output Class Reference

This widget is a subclass of Fl_Output that displays multiple lines of text.

`#include <Fl_Multiline_Output.H>`

Inheritance diagram for Fl_Multiline_Output:



Public Member Functions

- Fl_Multiline_Output (int X, int Y, int W, int H, const char ∗l=0)

  Creates a new Fl_Multiline_Output widget using the given position, size, and label string.

Additional Inherited Members

31.87.1 Detailed Description

This widget is a subclass of Fl_Output that displays multiple lines of text.

It also displays tab characters as whitespace to the next column.

Note that this widget does not support scrollbars, or per-character color control.

If you are presenting large amounts of read-only text and need scrollbars, or full color control of characters, then use Fl_Text_Display. If you want to display HTML text, use Fl_Help_View.

31.87.2 Constructor & Destructor Documentation

Fl_Multiline_Output::Fl_Multiline_Output (int *X*, int *Y*, int *W*, int *H*, const char ∗ *l* = `0`)

Creates a new Fl_Multiline_Output widget using the given position, size, and label string.

The default boxtype is FL_DOWN_BOX.

Inherited destructor destroys the widget and any value associated with it.

The documentation for this class was generated from the following files:

- Fl_Multiline_Output.H
- Fl_Input.cxx

31.88 Fl_Native_File_Chooser Class Reference

This class lets an FLTK application easily and consistently access the operating system's native file chooser.

`#include <Fl_Native_File_Chooser.H>`

Public Types

- enum Option {
  NO_OPTIONS = 0x0000, SAVEAS_CONFIRM = 0x0001, NEW_FOLDER = 0x0002, PREVIEW = 0x0004,
  USE_FILTER_EXT = 0x0008 }
- enum Type {
  BROWSE_FILE = 0, BROWSE_DIRECTORY, BROWSE_MULTI_FILE, BROWSE_MULTI_DIRECTORY,
  BROWSE_SAVE_FILE, BROWSE_SAVE_DIRECTORY }

Public Member Functions

- int count () const

  Returns the number of filenames (or directory names) the user selected.
- void directory (const char ∗val)

  Preset the directory the browser will show when opened.
- const char ∗ directory () const

  Returns the current preset directory() value.
- const char ∗ errmsg () const

  Returns a system dependent error message for the last method that failed.
- const char ∗ filename () const

  Return the filename the user chose.
- const char ∗ filename (int i) const

  Return one of the filenames the user selected.
- const char ∗ filter () const

  Returns the filter string last set.
- void filter (const char ∗f)

  Sets the filename filters used for browsing.
- void filter_value (int i)

  Sets which filter will be initially selected.
- int filter_value () const

  Returns which filter value was last selected by the user.
- int filters () const

Gets how many filters were available, not including "All Files".

- Fl_Native_File_Chooser (int val=BROWSE_FILE)

  The constructor.
- void options (int o)

  Sets the platform specific chooser options to `val`.
- int options () const

  Gets the platform specific Fl_Native_File_Chooser::Option flags.
- void preset_file (const char ∗f)

  Sets the default filename for the chooser.
- const char ∗ preset_file () const

  Get the preset filename.
- int show ()

  Post the chooser's dialog.
- void title (const char ∗t)

  Set the title of the file chooser's dialog window.
- const char ∗ title () const

  Get the title of the file chooser's dialog window.
- void type (int t)

  Sets the current Fl_Native_File_Chooser::Type of browser.
- int type () const

  Gets the current Fl_Native_File_Chooser::Type of browser.
- ∼Fl_Native_File_Chooser ()

  Destructor.

Static Public Attributes

- static const char ∗ file_exists_message = "File exists. Are you sure you want to overwrite?"

  Localizable message.

31.88.1 Detailed Description

This class lets an FLTK application easily and consistently access the operating system's native file chooser.

Some operating systems have very complex and specific file choosers that many users want access to specifically, instead of FLTK's default file chooser(s).

In cases where there is no native file browser, FLTK's own file browser is used instead.

To use this widget, use the following include in your code:

```
#include <FL/Fl_Native_File_Chooser.H>
```

The following example shows how to pick a single file:

```
// Create and post the local native file chooser
#include <FL/Fl_Native_File_Chooser.H>
[..]
Fl_Native_File_Chooser fnfc;
fnfc.title("Pick a file");
fnfc.type(Fl_Native_File_Chooser::BROWSE_FILE);
fnfc.filter("Text\t*.txt\n"
            "C Files\t*.{cxx,h,c}");
fnfc.directory("/var/tmp");           // default directory to use
// Show native chooser
switch ( fnfc.show() ) {
  case -1: printf("ERROR: %s\n", fnfc.errmsg());    break;  // ERROR
  case  1: printf("CANCEL\n");                      break;  // CANCEL
  default: printf("PICKED: %s\n", fnfc.filename()); break;  // FILE CHOSEN
}
```

The Fl_Native_File_Chooser widget transmits UTF-8 encoded filenames to its user. It is recommended to open files that may have non-ASCII names with the fl_fopen() or fl_open() utility functions that handle these names in a cross-platform way (whereas the standard fopen()/open() functions fail on the MS-Windows platform to open files with a non-ASCII name).

Platform Specific Caveats

- Under X windows, and if Fl::OPTION_FNFC_USES_GTK has not been switched off, the widget attempts to use standard GTK file chooser dialogs if they are available at run-time on the platform, and falls back to use FLTK's Fl_File_Chooser if they are not. In the latter case, it's best if you call Fl_File_Icon::load_system_icons() at the start of main(), to enable the nicer looking file browser widgets. Use the static public attributes of class Fl_File_Chooser to localize the browser.
- Some operating systems support certain OS specific options; see Fl_Native_File_Chooser::options() for a list.

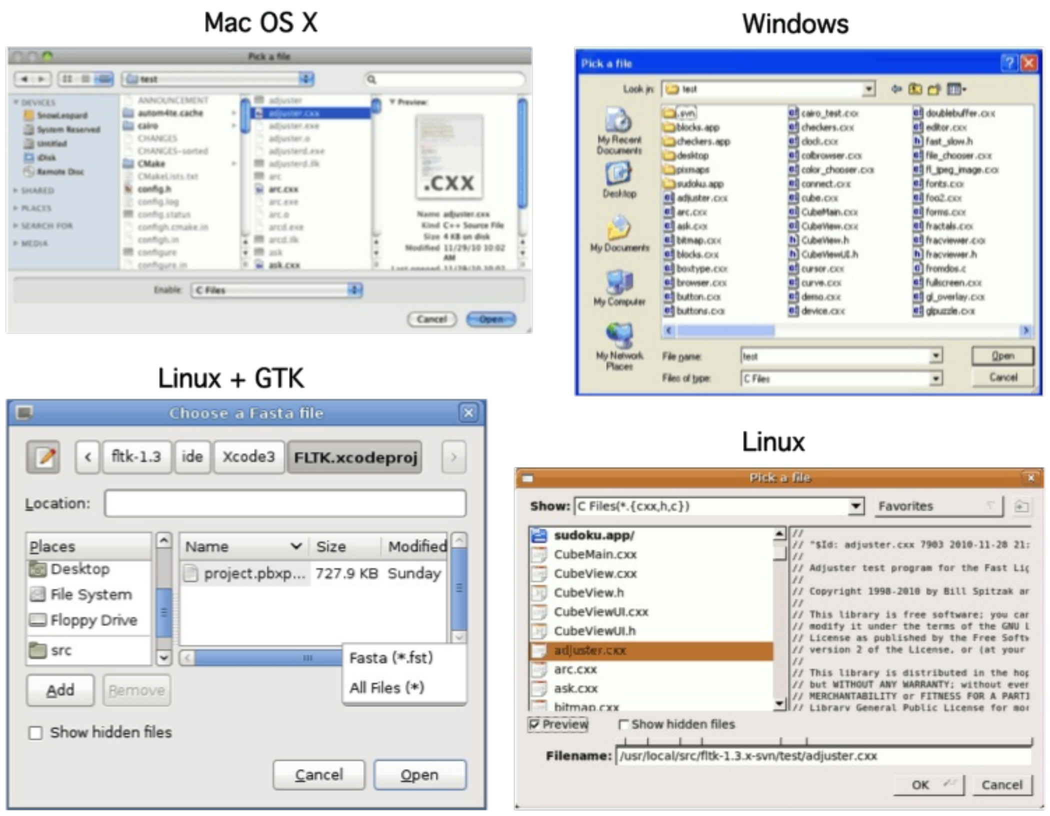


Figure 31.22: The Fl_Native_File_Chooser on different platforms

31.88.2 Member Enumeration Documentation

enum Fl_Native_File_Chooser::Option

Enumerator

NO_OPTIONS	no options enabled
SAVEAS_CONFIRM	Show native 'Save As' overwrite confirm dialog.
NEW_FOLDER	Show 'New Folder' icon (if supported)

PREVIEW enable preview mode (if supported)

USE_FILTER_EXT Chooser filter pilots the output file extension (if supported)

enum Fl_Native_File_Chooser::Type

Enumerator

BROWSE_FILE browse files (lets user choose one file)

BROWSE_DIRECTORY browse directories (lets user choose one directory)

BROWSE_MULTI_FILE browse files (lets user choose multiple files)

BROWSE_MULTI_DIRECTORY browse directories (lets user choose multiple directories)

BROWSE_SAVE_FILE browse to save a file

BROWSE_SAVE_DIRECTORY browse to save a directory

31.88.3 Constructor & Destructor Documentation

Fl_Native_File_Chooser::Fl_Native_File_Chooser (int *val* = BROWSE_FILE)

The constructor.

Internally allocates the native widgets. Optional `val` presets the type of browser this will be, which can also be changed with type().

Fl_Native_File_Chooser::∼Fl_Native_File_Chooser ( )

Destructor.

Deallocates any resources allocated to this widget.

31.88.4 Member Function Documentation

int Fl_Native_File_Chooser::count ( ) const

Returns the number of filenames (or directory names) the user selected.

Example:

```
if ( fnfc->show() == 0 ) {
    // Print all filenames user selected
    for (int n=0; n<fnfc->count(); n++ ) {
        printf("%d) '%s'\n", n, fnfc->filename(n));
    }
}
```

void Fl_Native_File_Chooser::directory (const char ∗ *val*)

Preset the directory the browser will show when opened.

If `val` is NULL, or no directory is specified, the chooser will attempt to use the last non-cancelled folder.

const char ∗ Fl_Native_File_Chooser::errmsg ( ) const

Returns a system dependent error message for the last method that failed.

This message should at least be flagged to the user in a dialog box, or to some kind of error log. Contents will be valid only for methods that document errmsg() will have info on failures.

const char ∗ Fl_Native_File_Chooser::filename ( ) const

Return the filename the user chose.

Use this if only expecting a single filename. If more than one filename is expected, use filename(int) instead. Return value may be "" if no filename was chosen (eg. user cancelled).

const char ∗ Fl_Native_File_Chooser::filename (int *i*) const

Return one of the filenames the user selected.

Use count() to determine how many filenames the user selected.

Example:

```
if ( fnfc->show() == 0 ) {
    // Print all filenames user selected
    for (int n=0; n<fnfc->count(); n++ ) {
        printf("%d) '%s'\n", n, fnfc->filename(n));
    }
}
```

const char ∗ Fl_Native_File_Chooser::filter ( ) const

Returns the filter string last set.

Can be NULL if no filter was set.

void Fl_Native_File_Chooser::filter (const char ∗ *f*)

Sets the filename filters used for browsing.

The default is NULL, which browses all files.

The filter string can be any of:

- A single wildcard (eg. "∗.txt")
- Multiple wildcards (eg. "∗.{cxx,h,H}")
- A descriptive name followed by a "\t" and a wildcard (eg. "Text Files\t∗.txt")
- A list of separate wildcards with a "\n" between each (eg. "∗.{cxx,H}\n∗.txt")
- A list of descriptive names and wildcards (eg. "C++ Files\t∗.{cxx,H}\nTxt Files\t∗.txt")

The format of each filter is a wildcard, or an optional user description followed by '\t' and the wildcard.

On most platforms, each filter is available to the user via a pulldown menu in the file chooser. The 'All Files' option is always available to the user.

void Fl_Native_File_Chooser::filter_value (int *i*)

Sets which filter will be initially selected.

The first filter is indexed as 0. If filter_value()==filters(), then "All Files" was chosen. If filter_value() > filters(), then a custom filter was set.

int Fl_Native_File_Chooser::filter_value ( ) const

Returns which filter value was last selected by the user.

This is only valid if the chooser returns success.

void Fl_Native_File_Chooser::options (int *o*)

Sets the platform specific chooser options to `val`.

`val` is expected to be one or more Fl_Native_File_Chooser::Option flags ORed together. Some platforms have OS-specific functions that can be enabled/disabled via this method.

```
Flag              Description                                       Win       Mac       Other
--------------    -----------------------------------------------   -------   -------   -------
NEW_FOLDER        Shows the 'New Folder' button.                    Ignored   Used      Used
PREVIEW           Enables the 'Preview' mode by default.            Ignored   Ignored   Used
SAVEAS_CONFIRM    Confirm dialog if BROWSE_SAVE_FILE file exists.   Used
       Used      Used
USE_FILTER_EXT    Chooser filter pilots the output file extension.  Ignored   Used
       Used (GTK)
```

void Fl_Native_File_Chooser::preset_file (const char ∗ *f*)

Sets the default filename for the chooser.

Use directory() to set the default directory. Mainly used to preset the filename for save dialogs, and on most platforms can be used for opening files as well.

int Fl_Native_File_Chooser::show ( )

Post the chooser's dialog.

Blocks until dialog has been completed or cancelled.

Returns

- 0 – user picked a file
- 1 – user cancelled
- -1 – failed; errmsg() has reason

void Fl_Native_File_Chooser::title (const char ∗ *t*)

Set the title of the file chooser's dialog window.

Can be NULL if no title desired. The default title varies according to the platform, so you are advised to set the title explicitly.

const char ∗ Fl_Native_File_Chooser::title ( ) const

Get the title of the file chooser's dialog window.

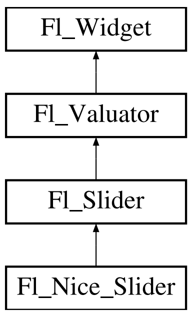
Return value may be NULL if no title was set.

The documentation for this class was generated from the following files:

- Fl_Native_File_Chooser.H
- Fl_Native_File_Chooser.cxx
- Fl_Native_File_Chooser_FLTK.cxx

31.89 Fl_Nice_Slider Class Reference

Inheritance diagram for Fl_Nice_Slider:



Public Member Functions

- **Fl_Nice_Slider** (int X, int Y, int W, int H, const char ∗L=0)

Additional Inherited Members

The documentation for this class was generated from the following files:

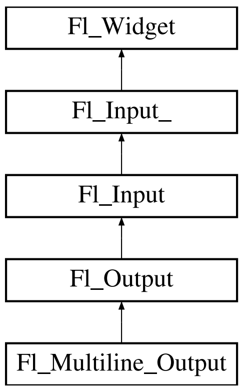
- Fl_Nice_Slider.H
- Fl_Slider.cxx

31.90 Fl_Output Class Reference

This widget displays a piece of text.

`#include <Fl_Output.H>`

Inheritance diagram for Fl_Output:



Public Member Functions

- Fl_Output (int X, int Y, int W, int H, const char ∗l=0)

  Creates a new Fl_Output widget using the given position, size, and label string.

Additional Inherited Members

31.90.1 Detailed Description

This widget displays a piece of text.

When you set the value() , Fl_Output does a strcpy() to its own storage, which is useful for program-generated values. The user may select portions of the text using the mouse and paste the contents into other fields or programs.

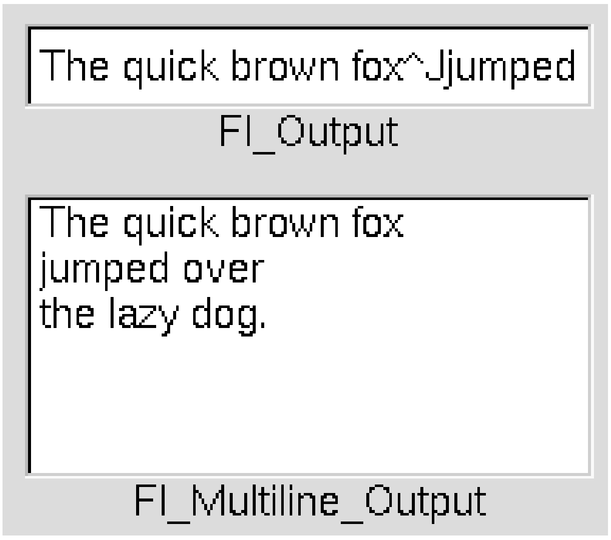


Figure 31.23: Fl_Output

There is a single subclass, Fl_Multiline_Output, which allows you to display multiple lines of text. Fl_Multiline_Output does not provide scroll bars. If a more complete text editing widget is needed, use Fl_Text_Display instead.

The text may contain any characters except \0, and will correctly display anything, using ^X notation for unprintable control characters and \nnn notation for unprintable characters with the high bit set. It assumes the font can draw any characters in the ISO-Latin1 character set.

31.90.2 Constructor & Destructor Documentation

Fl_Output::Fl_Output (int *X*, int *Y*, int *W*, int *H*, const char ∗ *l* = *0*)

Creates a new Fl_Output widget using the given position, size, and label string.

The default boxtype is FL_DOWN_BOX.

Inherited destructor destroys the widget and any value associated with it.

The documentation for this class was generated from the following files:

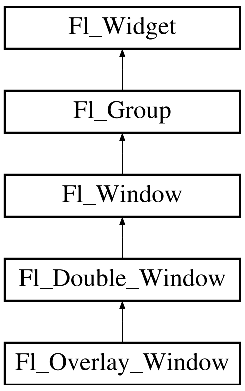
- Fl_Output.H
- Fl_Input.cxx

31.91 Fl_Overlay_Window Class Reference

This window provides double buffering and also the ability to draw the "overlay" which is another picture placed on top of the main image.

`#include <Fl_Overlay_Window.H>`

Inheritance diagram for Fl_Overlay_Window:



Public Member Functions

- int can_do_overlay ()

  Returns non-zero if there's hardware overlay support.
- void flush ()

  Forces the window to be redrawn.
- void hide ()

  Removes the window from the screen.
- void redraw_overlay ()

  Call this to indicate that the overlay data has changed and needs to be redrawn.
- void resize (int, int, int, int)

  Changes the size and position of the window.
- void show ()

  Puts the window on the screen.
- void **show** (int a, char ∗∗b)
- ∼Fl_Overlay_Window ()

  Destroys the window and all child widgets.

Protected Member Functions

- virtual void draw_overlay ()=0

  You must subclass Fl_Overlay_Window and provide this method.
- Fl_Overlay_Window (int W, int H, const char ∗l=0)

  See Fl_Overlay_Window::Fl_Overlay_Window(int X, int Y, int W, int H, const char ∗l=0)
- Fl_Overlay_Window (int X, int Y, int W, int H, const char ∗l=0)

  Creates a new Fl_Overlay_Window widget using the given position, size, and label (title) string.

Additional Inherited Members

31.91.1 Detailed Description

This window provides double buffering and also the ability to draw the "overlay" which is another picture placed on top of the main image.

The overlay is designed to be a rapidly-changing but simple graphic such as a mouse selection box. Fl_Overlay_Window uses the overlay planes provided by your graphics hardware if they are available.

If no hardware support is found the overlay is simulated by drawing directly into the on-screen copy of the double-buffered window, and "erased" by copying the backbuffer over it again. This means the overlay will blink if you change the image in the window.

31.91.2 Constructor & Destructor Documentation

Fl_Overlay_Window::Fl_Overlay_Window (int *X*, int *Y*, int *W*, int *H*, const char * *l* = `0`) `[protected]`

Creates a new Fl_Overlay_Window widget using the given position, size, and label (title) string.

If the positions (x,y) are not given, then the window manager will choose them.

31.91.3 Member Function Documentation

virtual void Fl_Overlay_Window::draw_overlay ( ) `[protected]`, `[pure virtual]`

You must subclass Fl_Overlay_Window and provide this method.

It is just like a draw() method, except it draws the overlay. The overlay will have already been "cleared" when this is called. You can use any of the routines described in <FL/fl_draw.H>.

void Fl_Overlay_Window::hide ( ) `[virtual]`

Removes the window from the screen.

If the window is already hidden or has not been shown then this does nothing and is harmless.

Reimplemented from Fl_Double_Window.

void Fl_Overlay_Window::redraw_overlay ( )

Call this to indicate that the overlay data has changed and needs to be redrawn.

The overlay will be clear until the first time this is called, so if you want an initial display you must call this after calling show().

void Fl_Overlay_Window::resize (int *X*, int *Y*, int *W*, int *H*) `[virtual]`

Changes the size and position of the window.

If shown() is true, these changes are communicated to the window server (which may refuse that size and cause a further resize). If shown() is false, the size and position are used when show() is called. See Fl_Group for the effect of resizing on the child widgets.

You can also call the Fl_Widget methods size(x,y) and position(w,h), which are inline wrappers for this virtual function.

A top-level window can not force, but merely suggest a position and size to the operating system. The window manager may not be willing or able to display a window at the desired position or with the given dimensions. It is up to the application developer to verify window parameters after the resize request.

Reimplemented from Fl_Double_Window.

void Fl_Overlay_Window::show ( ) `[virtual]`

Puts the window on the screen.

Usually (on X) this has the side effect of opening the display.

If the window is already shown then it is restored and raised to the top. This is really convenient because your program can call show() at any time, even if the window is already up. It also means that show() serves the purpose of raise() in other toolkits.

Fl_Window::show(int argc, char **argv) is used for top-level windows and allows standard arguments to be parsed from the command-line.

Note

For some obscure reasons Fl_Window::show() resets the current group by calling Fl_Group::current(0). The comments in the code say "get rid of very common user bug: forgot end()". Although this is true it may have unwanted side effects if you show() an unrelated window (maybe for an error message or warning) while building a window or any other group widget.

Todo Check if we can remove resetting the current group in a later FLTK version (after 1.3.x). This may break "already broken" programs though if they rely on this "feature".

See Also

Fl_Window::show(int argc, char **argv)

Reimplemented from Fl_Double_Window.
The documentation for this class was generated from the following files:

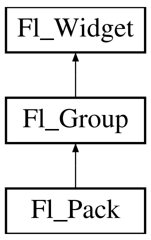
- Fl_Overlay_Window.H
- Fl_Double_Window.cxx
- Fl_Overlay_Window.cxx

31.92 Fl_Pack Class Reference

This widget was designed to add the functionality of compressing and aligning widgets.

`#include <Fl_Pack.H>`

Inheritance diagram for Fl_Pack:



Public Types

- enum { **VERTICAL** = 0, **HORIZONTAL** = 1 }

Public Member Functions

- Fl_Pack (int x, int y, int w, int h, const char *l=0)

  Creates a new Fl_Pack widget using the given position, size, and label string.
- uchar horizontal () const

  Same as Fl_Group::type()
- int spacing () const

  Gets the number of extra pixels of blank space that are added between the children.
- void spacing (int i)

  Sets the number of extra pixels of blank space that are added between the children.

Protected Member Functions

- void draw ()

  Draws the widget.

Additional Inherited Members

31.92.1 Detailed Description

This widget was designed to add the functionality of compressing and aligning widgets.

If type() is Fl_Pack::HORIZONTAL all the children are resized to the height of the Fl_Pack, and are moved next to each other horizontally. If type() is not Fl_Pack::HORIZONTAL then the children are resized to the width and are stacked below each other. Then the Fl_Pack resizes itself to surround the child widgets.

This widget is needed for the Fl_Tabs. In addition you may want to put the Fl_Pack inside an Fl_Scroll.

The resizable for Fl_Pack is set to NULL by default.

See also: Fl_Group::resizable()

31.92.2 Constructor & Destructor Documentation

Fl_Pack::Fl_Pack (int *X*, int *Y*, int *W*, int *H*, const char ∗ *l* = *0*)

Creates a new Fl_Pack widget using the given position, size, and label string.

The default boxtype is FL_NO_BOX.

The destructor *also deletes all the children*. This allows a whole tree to be deleted at once, without having to keep a pointer to all the children in the user code. A kludge has been done so the Fl_Pack and all of it's children can be automatic (local) variables, but you must declare the Fl_Pack*first*, so that it is destroyed last.

31.92.3 Member Function Documentation

void Fl_Pack::draw ( ) `[protected]`, `[virtual]`

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call redraw() instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw() method*, e.g. for an embedded scrollbar, you can do it (because draw() is virtual) like this:

```
Fl_Widget *s = &scroll;                // scroll is an embedded Fl_Scrollbar
s->draw();                   // calls Fl_Scrollbar::draw()
```

Reimplemented from Fl_Group.

The documentation for this class was generated from the following files:

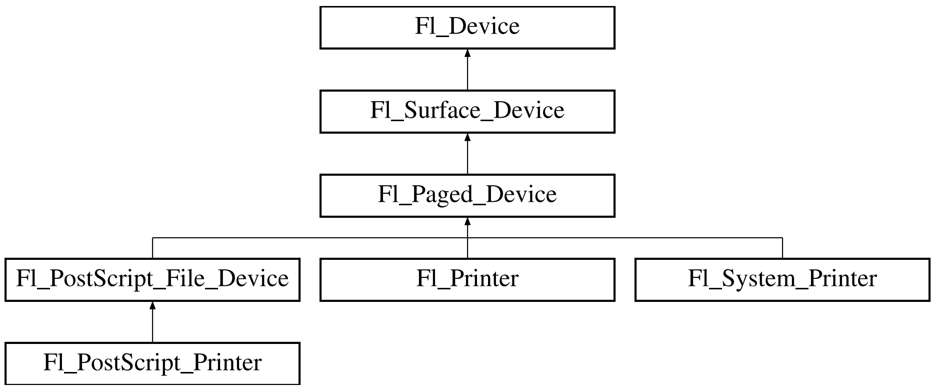
- Fl_Pack.H
- Fl_Pack.cxx

31.93 Fl_Paged_Device Class Reference

Represents page-structured drawing surfaces.

`#include <Fl_Paged_Device.H>`

Inheritance diagram for Fl_Paged_Device:



Classes

- struct page_format

  width, height and name of a page format

Public Types

- enum Page_Format {
  A0 = 0, **A1**, **A2**, **A3**,
  A4, **A5**, **A6**, **A7**,
  A8, **A9**, **B0**, **B1**,
  B2, **B3**, **B4**, **B5**,
  B6, **B7**, **B8**, **B9**,
  B10, **C5E**, **DLE**, **EXECUTIVE**,
  FOLIO, **LEDGER**, **LEGAL**, LETTER,
  TABLOID, **ENVELOPE**, **MEDIA** = 0x1000 }

  Possible page formats.
- enum Page_Layout { PORTRAIT = 0, LANDSCAPE = 0x100, REVERSED = 0x200, ORIENTATION = 0x300 }

  Possible page layouts.

Public Member Functions

- const char ∗ class_name ()

  Returns the name of the class of this object.
- virtual void end_job (void)

  To be called at the end of a print job.
- virtual int end_page (void)

  To be called at the end of each page.
- virtual void margins (int ∗left, int ∗top, int ∗right, int ∗bottom)

  Computes the dimensions of margins that lie between the printable page area and the full page.
- virtual void origin (int x, int y)

  Sets the position in page coordinates of the origin of graphics functions.
- virtual void origin (int ∗x, int ∗y)

  Computes the page coordinates of the current origin of graphics functions.
- virtual void print_widget (Fl_Widget ∗widget, int delta_x=0, int delta_y=0)

  Draws the widget on the printed page.

- void print_window (Fl_Window ∗win, int x_offset=0, int y_offset=0)

  Prints a window with its title bar and frame if any.
- virtual void print_window_part (Fl_Window ∗win, int x, int y, int w, int h, int delta_x=0, int delta_y=0)

  Prints a rectangular part of an on-screen window.
- virtual int printable_rect (int ∗w, int ∗h)

  Computes the width and height of the printable area of the page.
- virtual void rotate (float angle)

  Rotates the graphics operations relatively to paper.
- virtual void scale (float scale_x, float scale_y=0.)

  Changes the scaling of page coordinates.
- virtual int start_job (int pagecount, int ∗frompage=NULL, int ∗topage=NULL)

  Starts a print job.
- virtual int start_page (void)

  Starts a new printed page.
- virtual void translate (int x, int y)

  Translates the current graphics origin accounting for the current rotation.
- virtual void untranslate (void)

  Undoes the effect of a previous translate() call.
- virtual ∼Fl_Paged_Device ()

  The destructor.

Static Public Attributes

- static const char ∗ **class_id** = "Fl_Paged_Device"
- static const page_format page_formats [NO_PAGE_FORMATS]

  width, height and name of all elements of the enum Page_Format.

Protected Member Functions

- Fl_Paged_Device ()

  The constructor.

Protected Attributes

- int x_offset

  horizontal offset to the origin of graphics coordinates
- int y_offset

  vertical offset to the origin of graphics coordinates

Friends

- class **Fl_Copy_Surface**
- class **Fl_Image_Surface**

Additional Inherited Members

31.93.1 Detailed Description

Represents page-structured drawing surfaces.

This class has no public constructor: don't instantiate it; use Fl_Printer or Fl_PostScript_File_Device instead.

31.93.2 Member Enumeration Documentation

enum Fl_Paged_Device::Page_Format

Possible page formats.
All paper formats with pre-defined width and height.

Enumerator

A0 A0 format.

A4 A4 format.

LETTER Letter format.

enum Fl_Paged_Device::Page_Layout

Possible page layouts.

Enumerator

PORTRAIT Portrait orientation.

LANDSCAPE Landscape orientation.

REVERSED Reversed orientation.

ORIENTATION orientation

31.93.3 Member Function Documentation

const char∗ Fl_Paged_Device::class_name ( ) `[inline]`, `[virtual]`

Returns the name of the class of this object.
Use of the class_name() function is discouraged because it will be removed from future FLTK versions.
The class of an instance of an Fl_Device subclass can be checked with code such as:

```
if ( instance->class_name() == Fl_Printer::class_id ) { ... }
```

Reimplemented from Fl_Surface_Device.
Reimplemented in Fl_PostScript_File_Device, Fl_Printer, Fl_PostScript_Printer, and Fl_System_Printer.

int Fl_Paged_Device::end_page (void) `[virtual]`

To be called at the end of each page.

Returns

0 if OK, non-zero if any error.

Reimplemented in Fl_PostScript_File_Device, Fl_Printer, and Fl_System_Printer.

void Fl_Paged_Device::margins (int ∗ *left*, int ∗ *top*, int ∗ *right*, int ∗ *bottom*) `[virtual]`

Computes the dimensions of margins that lie between the printable page area and the full page.
Values are in the same unit as that used by FLTK drawing functions. They are changed by scale() calls.
Parameters

out	*left*	If non-null, ∗left is set to the left margin size.
out	*top*	If non-null, ∗top is set to the top margin size.
out	*right*	If non-null, ∗right is set to the right margin size.
out	*bottom*	If non-null, ∗bottom is set to the bottom margin size.

Reimplemented in Fl_PostScript_File_Device, Fl_Printer, and Fl_System_Printer.

void Fl_Paged_Device::origin (int *x*, int *y*) `[virtual]`

Sets the position in page coordinates of the origin of graphics functions.

Arguments should be expressed relatively to the result of a previous printable_rect() call. That is, `printable_rect(&w, &h); origin(w/2, 0);` sets the graphics origin at the top center of the page printable area. Origin() calls are not affected by rotate() calls. Successive origin() calls don't combine their effects.

Parameters

in	*x*	Horizontal position in page coordinates of the desired origin of graphics functions.
in	*y*	Same as above, vertically.

Reimplemented in Fl_PostScript_File_Device, Fl_Printer, and Fl_System_Printer.

void Fl_Paged_Device::origin (int ∗ *x*, int ∗ *y*) `[virtual]`

Computes the page coordinates of the current origin of graphics functions.

Parameters

out	*x*	If non-null, ∗x is set to the horizontal page offset of graphics origin.
out	*y*	Same as above, vertically.

Reimplemented in Fl_PostScript_File_Device, Fl_Printer, and Fl_System_Printer.

void Fl_Paged_Device::print_widget (Fl_Widget ∗ *widget*, int *delta_x* = *0*, int *delta_y* = *0*) `[virtual]`

Draws the widget on the printed page.

The widget's position on the printed page is determined by the last call to origin() and by the optional delta_x and delta_y arguments. Its dimensions are in points unless there was a previous call to scale().

Parameters

in	*widget*	Any FLTK widget (e.g., standard, custom, window).
in	*delta_x*	Optional horizontal offset for positioning the widget relatively to the current origin of graphics functions.
in	*delta_y*	Same as above, vertically.

Reimplemented in Fl_Printer.

void Fl_Paged_Device::print_window (Fl_Window ∗ *win*, int *x_offset* = *0*, int *y_offset* = *0*)

Prints a window with its title bar and frame if any.

`x_offset` and `y_offset` are optional coordinates of where to position the window top left. Equivalent to print_widget() if `win` is a subwindow or has no border. Use Fl_Window::decorated_w() and Fl_Window::decorated_h() to get the size of the printed window.

void Fl_Paged_Device::print_window_part (Fl_Window ∗ *win*, int *x*, int *y*, int *w*, int *h*, int *delta_x* = *0*, int *delta_y* = *0*) `[virtual]`

Prints a rectangular part of an on-screen window.

Parameters

win	The window from where to capture.
x	The rectangle left
y	The rectangle top
w	The rectangle width
h	The rectangle height
delta_x	Optional horizontal offset from current graphics origin where to print the captured rectangle.
delta_y	As above, vertically.

Reimplemented in Fl_Printer.

int Fl_Paged_Device::printable_rect (int ∗ *w*, int ∗ *h*) `[virtual]`

Computes the width and height of the printable area of the page.

Values are in the same unit as that used by FLTK drawing functions, are unchanged by calls to origin(), but are changed by scale() calls. Values account for the user-selected paper type and print orientation.

Returns

0 if OK, non-zero if any error

Reimplemented in Fl_PostScript_File_Device, Fl_Printer, and Fl_System_Printer.

void Fl_Paged_Device::rotate (float *angle*) `[virtual]`

Rotates the graphics operations relatively to paper.

The rotation is centered on the current graphics origin. Successive rotate() calls don't combine their effects.

Parameters

angle	Rotation angle in counter-clockwise degrees.

Reimplemented in Fl_PostScript_File_Device, Fl_Printer, and Fl_System_Printer.

void Fl_Paged_Device::scale (float *scale_x*, float *scale_y* = `0.`) `[virtual]`

Changes the scaling of page coordinates.

This function also resets the origin of graphics functions at top left of printable page area. After a scale() call, do a printable_rect() call to get the new dimensions of the printable page area. Successive scale() calls don't combine their effects.

Parameters

scale_x	Horizontal dimensions of plot are multiplied by this quantity.
scale_y	Same as above, vertically. The value 0. is equivalent to setting `scale_y` = `scale_x`. Thus, scale(factor); is equivalent to scale(factor, factor);

Reimplemented in Fl_PostScript_File_Device, Fl_Printer, and Fl_System_Printer.

int Fl_Paged_Device::start_job (int *pagecount*, int ∗ *frompage* = `NULL`, int ∗ *topage* = `NULL`) `[virtual]`

Starts a print job.

Parameters

in	*pagecount*	the total number of pages of the job (or 0 if you don't know the number of pages)
out	*frompage*	if non-null, ∗frompage is set to the first page the user wants printed
out	*topage*	if non-null, ∗topage is set to the last page the user wants printed

Returns

0 if OK, non-zero if any error

Reimplemented in Fl_PostScript_File_Device, Fl_Printer, Fl_PostScript_Printer, and Fl_System_Printer.

int Fl_Paged_Device::start_page (void) `[virtual]`

Starts a new printed page.

The page coordinates are initially in points, i.e., 1/72 inch, and with origin at the top left of the printable page area.

Returns

0 if OK, non-zero if any error

Reimplemented in Fl_PostScript_File_Device, Fl_Printer, and Fl_System_Printer.

void Fl_Paged_Device::translate (int *x*, int *y*) `[virtual]`

Translates the current graphics origin accounting for the current rotation.

This function is only useful after a rotate() call. Each translate() call must be matched by an untranslate() call. Successive translate() calls add up their effects.

Reimplemented in Fl_PostScript_File_Device, Fl_Printer, and Fl_System_Printer.

The documentation for this class was generated from the following files:

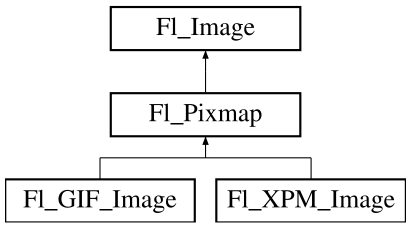
- Fl_Paged_Device.H
- Fl_Paged_Device.cxx

31.94 Fl_Pixmap Class Reference

The Fl_Pixmap class supports caching and drawing of colormap (pixmap) images, including transparency.

`#include <Fl_Pixmap.H>`

Inheritance diagram for Fl_Pixmap:



Public Member Functions

- virtual void color_average (Fl_Color c, float i)

    The color_average() method averages the colors in the image with the FLTK color value c.
- virtual Fl_Image ∗ copy (int W, int H)

    The copy() method creates a copy of the specified image.
- Fl_Image ∗ **copy** ()
- virtual void desaturate ()

    The desaturate() method converts an image to grayscale.
- virtual void draw (int X, int Y, int W, int H, int cx=0, int cy=0)

    Draws the image with a bounding box.
- void **draw** (int X, int Y)
- Fl_Pixmap (char ∗const ∗D)

    The constructors create a new pixmap from the specified XPM data.
- Fl_Pixmap (uchar ∗const ∗D)

    The constructors create a new pixmap from the specified XPM data.
- Fl_Pixmap (const char ∗const ∗D)

    The constructors create a new pixmap from the specified XPM data.
- Fl_Pixmap (const uchar ∗const ∗D)

    The constructors create a new pixmap from the specified XPM data.
- virtual void label (Fl_Widget ∗w)

    The label() methods are an obsolete way to set the image attribute of a widget or menu item.
- virtual void label (Fl_Menu_Item ∗m)

    The label() methods are an obsolete way to set the image attribute of a widget or menu item.
- virtual void uncache ()

    If the image has been cached for display, delete the cache data.
- virtual ∼Fl_Pixmap ()

    The destructor frees all memory and server resources that are used by the pixmap.

Public Attributes

- int **alloc_data**

Protected Member Functions

- void **measure** ()

Friends

- class **Fl_GDI_Graphics_Driver**
- class **Fl_GDI_Printer_Graphics_Driver**
- class **Fl_Quartz_Graphics_Driver**
- class **Fl_Xlib_Graphics_Driver**

Additional Inherited Members

31.94.1 Detailed Description

The Fl_Pixmap class supports caching and drawing of colormap (pixmap) images, including transparency.

31.94.2 Constructor & Destructor Documentation

Fl_Pixmap::Fl_Pixmap (char *const * *D*) `[inline]`, `[explicit]`

The constructors create a new pixmap from the specified XPM data.

Fl_Pixmap::Fl_Pixmap (uchar *const * *D*) `[inline]`, `[explicit]`

The constructors create a new pixmap from the specified XPM data.

Fl_Pixmap::Fl_Pixmap (const char *const * *D*) `[inline]`, `[explicit]`

The constructors create a new pixmap from the specified XPM data.

Fl_Pixmap::Fl_Pixmap (const uchar *const * *D*) `[inline]`, `[explicit]`

The constructors create a new pixmap from the specified XPM data.

31.94.3 Member Function Documentation

void Fl_Pixmap::color_average (Fl_Color *c*, float *i*) `[virtual]`

The color_average() method averages the colors in the image with the FLTK color value c.

The i argument specifies the amount of the original image to combine with the color, so a value of 1.0 results in no color blend, and a value of 0.0 results in a constant image of the specified color.

An internal copy is made of the original image before changes are applied, to avoid modifying the original image.

Reimplemented from Fl_Image.

Fl_Image * Fl_Pixmap::copy (int *W*, int *H*) `[virtual]`

The copy() method creates a copy of the specified image.

If the width and height are provided, the image is resized to the specified size. The image should be deleted (or in the case of Fl_Shared_Image, released) when you are done with it.

Reimplemented from Fl_Image.

void Fl_Pixmap::desaturate ( ) `[virtual]`

The desaturate() method converts an image to grayscale.

If the image contains an alpha channel (depth = 4), the alpha channel is preserved.

An internal copy is made of the original image before changes are applied, to avoid modifying the original image.

Reimplemented from Fl_Image.

void Fl_Pixmap::draw (int *X*, int *Y*, int *W*, int *H*, int *cx* = *0*, int *cy* = *0*) `[virtual]`

Draws the image with a bounding box.

Arguments `X,Y,W,H` specify a bounding box for the image, with the origin (upper-left corner) of the image offset by the `cx` and `cy` arguments.

In other words: `fl_push_clip(X,Y,W,H)` is applied, the image is drawn with its upper-left corner at `X-cx,Y-cy` and its own width and height, `fl_pop_clip()` is applied.

Reimplemented from Fl_Image.

void Fl_Pixmap::label (Fl_Widget ∗ *widget*) `[virtual]`

The label() methods are an obsolete way to set the image attribute of a widget or menu item.

Use the image() or deimage() methods of the Fl_Widget and Fl_Menu_Item classes instead.

Reimplemented from Fl_Image.

void Fl_Pixmap::label (Fl_Menu_Item ∗ *m*) `[virtual]`

The label() methods are an obsolete way to set the image attribute of a widget or menu item.

Use the image() or deimage() methods of the Fl_Widget and Fl_Menu_Item classes instead.

Reimplemented from Fl_Image.

void Fl_Pixmap::uncache ( ) `[virtual]`

If the image has been cached for display, delete the cache data.

This allows you to change the data used for the image and then redraw it without recreating an image object.

Reimplemented from Fl_Image.

The documentation for this class was generated from the following files:

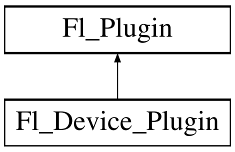
- Fl_Pixmap.H
- Fl_Pixmap.cxx

31.95 Fl_Plugin Class Reference

Fl_Plugin allows link-time and run-time integration of binary modules.

`#include <Fl_Plugin.H>`

Inheritance diagram for Fl_Plugin:



Public Member Functions

- Fl_Plugin (const char ∗klass, const char ∗name)

  Create a plugin.
- virtual ∼Fl_Plugin ()

  Clear the plugin and remove it from the database.

31.95.1 Detailed Description

Fl_Plugin allows link-time and run-time integration of binary modules.

Fl_Plugin and Fl_Plugin_Manager provide a small and simple solution for linking C++ classes at run-time, or optionally linking modules at compile time without the need to change the main application.

Fl_Plugin_Manager uses static initialisation to create the plugin interface early during startup. Plugins are stored in a temporary database, organized in classes.

Plugins should derive a new class from Fl_Plugin as a base:

```
class My_Plugin : public Fl_Plugin {
public:
  My_Plugin() : Fl_Plugin("effects", "blur") { }
  void do_something(...);
};
My_Plugin blur_plugin();
```

Plugins can be put into modules and either linked before distribution, or loaded from dynamically linkable files. An Fl_Plugin_Manager is used to list and access all currently loaded plugins.

```
Fl_Plugin_Manager mgr("effects");
int i, n = mgr.plugins();
for (i=0; i<n; i++) {
  My_Plugin *pin = (My_Plugin*)mgr.plugin(i);
  pin->do_something();
}
```

31.95.2 Constructor & Destructor Documentation

Fl_Plugin::Fl_Plugin (const char ∗ *klass*, const char ∗ *name*)

Create a plugin.
Parameters

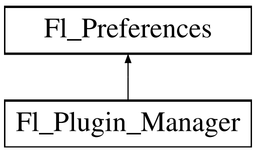
in	*klass*	plugins are grouped in classes
in	*name*	every plugin should have a unique name

The documentation for this class was generated from the following files:

- Fl_Plugin.H
- Fl_Preferences.cxx

31.96 Fl_Plugin_Manager Class Reference

Fl_Plugin_Manager manages link-time and run-time plugin binaries.
`#include <Fl_Plugin.H>`
Inheritance diagram for Fl_Plugin_Manager:



Public Member Functions

- Fl_Preferences::ID addPlugin (const char ∗name, Fl_Plugin ∗plugin)

  This function adds a new plugin to the database.
- Fl_Plugin_Manager (const char ∗klass)

  Manage all plugins belonging to one class.
- Fl_Plugin ∗ plugin (int index)

  Return the address of a plugin by index.
- Fl_Plugin ∗ plugin (const char ∗name)

  Return the address of a plugin by name.
- int plugins ()

  Return the number of plugins in the klass.
- ∼Fl_Plugin_Manager ()

  Remove the plugin manager.

Static Public Member Functions

- static int load (const char ∗filename)

  Load a module from disk.
- static int loadAll (const char ∗filepath, const char ∗pattern=0)

  Use this function to load a whole directory full of modules.
- static void removePlugin (Fl_Preferences::ID id)

  Remove any plugin.

Additional Inherited Members

31.96.1 Detailed Description

Fl_Plugin_Manager manages link-time and run-time plugin binaries.

See Also

Fl_Plugin

31.96.2 Constructor & Destructor Documentation

Fl_Plugin_Manager::∼Fl_Plugin_Manager ( )

Remove the plugin manager.

Calling this does not remove the database itself or any plugins. It just removes the reference to the database.

31.96.3 Member Function Documentation

Fl_Preferences::ID Fl_Plugin_Manager::addPlugin (const char ∗ *name,* Fl_Plugin ∗ *plugin*)

This function adds a new plugin to the database.

There is no need to call this function explicitly. Every Fl_Plugin constructor will call this function at initialization time.

int Fl_Plugin_Manager::load (const char ∗ *filename*) `[static]`

Load a module from disk.

A module must be a dynamically linkable file for the given operating system. When loading a module, its +init function will be called which in turn calls the constructor of all statically initialized Fl_Plugin classes and adds them to the database.

void Fl_Plugin_Manager::removePlugin (Fl_Preferences::ID *id*) `[static]`

Remove any plugin.

There is no need to call this function explicitly. Every Fl_Plugin destructor will call this function at destruction time.

The documentation for this class was generated from the following files:

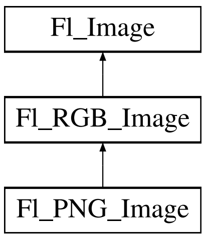
- Fl_Plugin.H
- Fl_Preferences.cxx

31.97 Fl_PNG_Image Class Reference

The Fl_PNG_Image class supports loading, caching, and drawing of Portable Network Graphics (PNG) image files.

`#include <Fl_PNG_Image.H>`

Inheritance diagram for Fl_PNG_Image:



Public Member Functions

- Fl_PNG_Image (const char ∗filename)

  The constructor loads the named PNG image from the given png filename.
- Fl_PNG_Image (const char ∗name_png, const unsigned char ∗buffer, int datasize)

  Constructor that reads a PNG image from memory.

Additional Inherited Members

31.97.1 Detailed Description

The Fl_PNG_Image class supports loading, caching, and drawing of Portable Network Graphics (PNG) image files.

The class loads colormapped and full-color images and handles color- and alpha-based transparency.

31.97.2 Constructor & Destructor Documentation

Fl_PNG_Image::Fl_PNG_Image (const char ∗ *filename*)

The constructor loads the named PNG image from the given png filename.

The destructor frees all memory and server resources that are used by the image.

Use Fl_Image::fail() to check if Fl_PNG_Image failed to load. fail() returns ERR_FILE_ACCESS if the file could not be opened or read, ERR_FORMAT if the PNG format could not be decoded, and ERR_NO_IMAGE if the image could not be loaded for another reason.

Parameters

in	*filename*	Name of PNG file to read

Fl_PNG_Image::Fl_PNG_Image (const char ∗ *name_png,* const unsigned char ∗ *buffer,* int *maxsize*)

Constructor that reads a PNG image from memory.

Construct an image from a block of memory inside the application. Fluid offers ”binary Data” chunks as a great way to add image data into the C++ source code. name_png can be NULL. If a name is given, the image is added to the list of shared images (see: Fl_Shared_Image) and will be available by that name.

Parameters

name_png	A name given to this image or NULL
buffer	Pointer to the start of the PNG image in memory
maxsize	Size in bytes of the memory buffer containing the PNG image

The documentation for this class was generated from the following files:

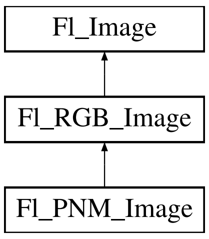
- Fl_PNG_Image.H
- Fl_PNG_Image.cxx

31.98 Fl_PNM_Image Class Reference

The Fl_PNM_Image class supports loading, caching, and drawing of Portable Anymap (PNM, PBM, PGM, PPM) image files.

`#include <Fl_PNM_Image.H>`

Inheritance diagram for Fl_PNM_Image:



Public Member Functions

- Fl_PNM_Image (const char ∗filename)

  The constructor loads the named PNM image.

Additional Inherited Members

31.98.1 Detailed Description

The Fl_PNM_Image class supports loading, caching, and drawing of Portable Anymap (PNM, PBM, PGM, PPM) image files.

The class loads bitmap, grayscale, and full-color images in both ASCII and binary formats.

31.98.2 Constructor & Destructor Documentation

Fl_PNM_Image::Fl_PNM_Image (const char ∗ *name*)

The constructor loads the named PNM image.

The destructor frees all memory and server resources that are used by the image.

Use Fl_Image::fail() to check if Fl_PNM_Image failed to load. fail() returns ERR_FILE_ACCESS if the file could not be opened or read, ERR_FORMAT if the PNM format could not be decoded, and ERR_NO_IMAGE if the image could not be loaded for another reason.

The documentation for this class was generated from the following files:

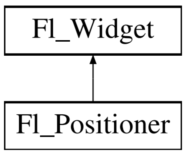
- Fl_PNM_Image.H
- Fl_PNM_Image.cxx

31.99 Fl_Positioner Class Reference

This class is provided for Forms compatibility.

```
#include <Fl_Positioner.H>
```

Inheritance diagram for Fl_Positioner:



Public Member Functions

- Fl_Positioner (int x, int y, int w, int h, const char ∗l=0)

  Creates a new Fl_Positioner widget using the given position, size, and label string.
- int handle (int)

  Handles the specified event.
- int value (double, double)

  Returns the current position in x and y.
- void xbounds (double, double)

  Sets the X axis bounds.
- double xmaximum () const

  Gets the X axis maximum.
- void xmaximum (double a)

  Same as xbounds(xminimum(), a)
- double xminimum () const

  Gets the X axis minimum.
- void xminimum (double a)

  Same as xbounds(a, xmaximum())
- void xstep (double a)

  Sets the stepping value for the X axis.
- double xvalue () const

  Gets the X axis coordinate.
- int xvalue (double)

  Sets the X axis coordinate.
- void ybounds (double, double)

  Sets the Y axis bounds.
- double ymaximum () const

  Gets the Y axis maximum.
- void ymaximum (double a)

  Same as ybounds(ymininimum(), a)
- double yminimum () const

  Gets the Y axis minimum.
- void yminimum (double a)

  Same as ybounds(a, ymaximum())
- void ystep (double a)

  Sets the stepping value for the Y axis.

- double yvalue () const

  Gets the Y axis coordinate.
- int yvalue (double)

  Sets the Y axis coordinate.

Protected Member Functions

- void **draw** (int, int, int, int)
- void draw ()

  Draws the widget.
- int **handle** (int, int, int, int, int)

Additional Inherited Members

31.99.1 Detailed Description

This class is provided for Forms compatibility.

It provides 2D input. It would be useful if this could be put atop another widget so that the crosshairs are on top, but this is not implemented. The color of the crosshairs is selection_color().

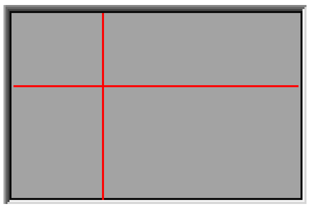


Figure 31.24: Fl_Positioner

31.99.2 Constructor & Destructor Documentation

Fl_Positioner::Fl_Positioner (int *X*, int *Y*, int *W*, int *H*, const char ∗ *l* = 0)

Creates a new Fl_Positioner widget using the given position, size, and label string.

The default boxtype is FL_NO_BOX.

31.99.3 Member Function Documentation

void Fl_Positioner::draw ( ) `[protected]`, `[virtual]`

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call redraw() instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw() method*, e.g. for an embedded scrollbar, you can do it (because draw() is virtual) like this:

```
Fl_Widget *s = &scroll;              // scroll is an embedded Fl_Scrollbar
s->draw();                      // calls Fl_Scrollbar::draw()
```

Implements Fl_Widget.

int Fl_Positioner::handle (int *event*) `[virtual]`

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited handle() method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters

`in`	*event*	the kind of event received

Return values

0	if the event was not used or understood
1	if the event was used and can be deleted

See Also

Fl_Event

Reimplemented from Fl_Widget.

int Fl_Positioner::value (double *X*, double *Y*)

Returns the current position in x and y.

void Fl_Positioner::xbounds (double *a*, double *b*)

Sets the X axis bounds.

void Fl_Positioner::xstep (double *a*) `[inline]`

Sets the stepping value for the X axis.

double Fl_Positioner::xvalue ( ) const `[inline]`

Gets the X axis coordinate.

int Fl_Positioner::xvalue (double *X*)

Sets the X axis coordinate.

void Fl_Positioner::ybounds (double *a*, double *b*)

Sets the Y axis bounds.

void Fl_Positioner::ystep (double *a*) `[inline]`

Sets the stepping value for the Y axis.

double Fl_Positioner::yvalue ( ) const `[inline]`

Gets the Y axis coordinate.

int Fl_Positioner::yvalue (double *Y*)

Sets the Y axis coordinate.

The documentation for this class was generated from the following files:

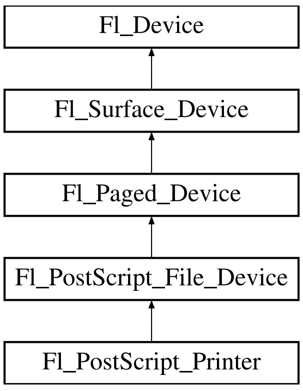
- Fl_Positioner.H
- Fl_Positioner.cxx

31.100 Fl_PostScript_File_Device Class Reference

To send graphical output to a PostScript file.

`#include <Fl_PostScript.H>`

Inheritance diagram for Fl_PostScript_File_Device:



Public Member Functions

- const char ∗ class_name ()

  Returns the name of the class of this object.
- void end_job (void)

  To be called at the end of a print job.
- int end_page (void)

  To be called at the end of each page.
- Fl_PostScript_File_Device ()

  The constructor.
- void margins (int ∗left, int ∗top, int ∗right, int ∗bottom)

  Computes the dimensions of margins that lie between the printable page area and the full page.
- void origin (int ∗x, int ∗y)

  Computes the page coordinates of the current origin of graphics functions.
- void origin (int x, int y)

  Sets the position in page coordinates of the origin of graphics functions.
- int printable_rect (int ∗w, int ∗h)

  Computes the width and height of the printable area of the page.
- void rotate (float angle)

  Rotates the graphics operations relatively to paper.
- void scale (float scale_x, float scale_y=0.)

  Changes the scaling of page coordinates.
- int start_job (int pagecount, int ∗from, int ∗to)

Don't use with this class.

- int start_job (int pagecount, enum Fl_Paged_Device::Page_Format format=Fl_Paged_Device::A4, enum Fl_Paged_Device::Page_Layout layout=Fl_Paged_Device::PORTRAIT)

  Begins the session where all graphics requests will go to a local PostScript file.
- int start_job (FILE ∗ps_output, int pagecount, enum Fl_Paged_Device::Page_Format format=Fl_Paged_Device::A4, enum Fl_Paged_Device::Page_Layout layout=Fl_Paged_Device::PORTRAIT)

  Begins the session where all graphics requests will go to FILE pointer.
- int start_page (void)

  Starts a new printed page.
- void translate (int x, int y)

  Translates the current graphics origin accounting for the current rotation.
- void untranslate (void)

  Undoes the effect of a previous translate() call.
- ∼Fl_PostScript_File_Device ()

  The destructor.

Static Public Attributes

- static const char ∗ **class_id** = "Fl_PostScript_File_Device"
- static const char ∗ file_chooser_title = "Select a .ps file"

  Label of the PostScript file chooser window.

Protected Member Functions

- Fl_PostScript_Graphics_Driver ∗ driver ()

  Returns the PostScript driver of this drawing surface.

Additional Inherited Members

31.100.1 Detailed Description

To send graphical output to a PostScript file.

This class is used exactly as the Fl_Printer class except for the start_job() call, two variants of which are usable and allow to specify what page format and layout are desired.

31.100.2 Member Function Documentation

const char∗ Fl_PostScript_File_Device::class_name ( ) [inline], [virtual]

Returns the name of the class of this object.

Use of the class_name() function is discouraged because it will be removed from future FLTK versions.

The class of an instance of an Fl_Device subclass can be checked with code such as:

```
if ( instance->class_name() == Fl_Printer::class_id ) { ... }
```

Reimplemented from Fl_Paged_Device.

Reimplemented in Fl_PostScript_Printer.

int Fl_PostScript_File_Device::end_page (void) [virtual]

To be called at the end of each page.

Returns

0 if OK, non-zero if any error.

Reimplemented from Fl_Paged_Device.

void Fl_PostScript_File_Device::margins (int ∗ *left*, int ∗ *top*, int ∗ *right*, int ∗ *bottom*) `[virtual]`

Computes the dimensions of margins that lie between the printable page area and the full page.

Values are in the same unit as that used by FLTK drawing functions. They are changed by scale() calls.

Parameters

`out`	*left*	If non-null, ∗left is set to the left margin size.
`out`	*top*	If non-null, ∗top is set to the top margin size.
`out`	*right*	If non-null, ∗right is set to the right margin size.
`out`	*bottom*	If non-null, ∗bottom is set to the bottom margin size.

Reimplemented from Fl_Paged_Device.

void Fl_PostScript_File_Device::origin (int ∗ *x*, int ∗ *y*) `[virtual]`

Computes the page coordinates of the current origin of graphics functions.

Parameters

`out`	*x*	If non-null, ∗x is set to the horizontal page offset of graphics origin.
`out`	*y*	Same as above, vertically.

Reimplemented from Fl_Paged_Device.

void Fl_PostScript_File_Device::origin (int *x*, int *y*) `[virtual]`

Sets the position in page coordinates of the origin of graphics functions.

Arguments should be expressed relatively to the result of a previous printable_rect() call. That is, `printable_rect(&w, &h); origin(w/2, 0);` sets the graphics origin at the top center of the page printable area. Origin() calls are not affected by rotate() calls. Successive origin() calls don't combine their effects.

Parameters

`in`	*x*	Horizontal position in page coordinates of the desired origin of graphics functions.
`in`	*y*	Same as above, vertically.

Reimplemented from Fl_Paged_Device.

int Fl_PostScript_File_Device::printable_rect (int ∗ *w*, int ∗ *h*) `[virtual]`

Computes the width and height of the printable area of the page.

Values are in the same unit as that used by FLTK drawing functions, are unchanged by calls to origin(), but are changed by scale() calls. Values account for the user-selected paper type and print orientation.

Returns

0 if OK, non-zero if any error

Reimplemented from Fl_Paged_Device.

void Fl_PostScript_File_Device::rotate (float *angle*) `[virtual]`

Rotates the graphics operations relatively to paper.

The rotation is centered on the current graphics origin. Successive rotate() calls don't combine their effects.

Parameters

angle	Rotation angle in counter-clockwise degrees.

Reimplemented from Fl_Paged_Device.

void Fl_PostScript_File_Device::scale (float *scale_x*, float *scale_y* = `0.`) `[virtual]`

Changes the scaling of page coordinates.

This function also resets the origin of graphics functions at top left of printable page area. After a scale() call, do a printable_rect() call to get the new dimensions of the printable page area. Successive scale() calls don't combine their effects.

Parameters

scale_x	Horizontal dimensions of plot are multiplied by this quantity.
scale_y	Same as above, vertically. The value 0. is equivalent to setting `scale_y = scale_x`. Thus, scale(factor); is equivalent to scale(factor, factor);

Reimplemented from Fl_Paged_Device.

int Fl_PostScript_File_Device::start_job (int *pagecount*, int ∗ *from*, int ∗ *to*) `[virtual]`

Don't use with this class.

Reimplemented from Fl_Paged_Device.

Reimplemented in Fl_PostScript_Printer.

int Fl_PostScript_File_Device::start_job (int *pagecount*, enum Fl_Paged_Device::Page_Format *format* = Fl_Paged_Device::A4, enum Fl_Paged_Device::Page_Layout *layout* = Fl_Paged_Device::PORTRAIT)

Begins the session where all graphics requests will go to a local PostScript file.

Opens a file dialog entitled with Fl_PostScript_File_Device::file_chooser_title to select an output PostScript file.

Parameters

pagecount	The total number of pages to be created. Use 0 if this number is unknown when this function is called.
format	Desired page format.
layout	Desired page layout.

Returns

0 if OK, 1 if user cancelled the file dialog, 2 if fopen failed on user-selected output file.

int Fl_PostScript_File_Device::start_job (FILE ∗ *ps_output*, int *pagecount*, enum Fl_Paged_Device::Page_Format *format* = Fl_Paged_Device::A4, enum Fl_Paged_Device::Page_Layout *layout* = Fl_Paged_Device::PORTRAIT)

Begins the session where all graphics requests will go to FILE pointer.

Parameters

ps_output	A writable FILE pointer that will receive PostScript output and that should not be closed until after end_job() has been called.

pagecount	The total number of pages to be created. Use 0 if this number is unknown when this function is called.
format	Desired page format.
layout	Desired page layout.

Returns

always 0.

int Fl_PostScript_File_Device::start_page (void) `[virtual]`

Starts a new printed page.

The page coordinates are initially in points, i.e., 1/72 inch, and with origin at the top left of the printable page area.

Returns

0 if OK, non-zero if any error

Reimplemented from Fl_Paged_Device.

void Fl_PostScript_File_Device::translate (int *x*, int *y*) `[virtual]`

Translates the current graphics origin accounting for the current rotation.

This function is only useful after a rotate() call. Each translate() call must be matched by an untranslate() call. Successive translate() calls add up their effects.

Reimplemented from Fl_Paged_Device.

The documentation for this class was generated from the following files:

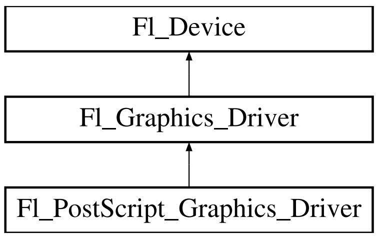
- Fl_PostScript.H
- Fl_PostScript.cxx

31.101 Fl_PostScript_Graphics_Driver Class Reference

PostScript graphical backend.

`#include <Fl_PostScript.H>`

Inheritance diagram for Fl_PostScript_Graphics_Driver:



Public Member Functions

- void arc (double x, double y, double r, double start, double a)

  see fl_arc(double x, double y, double r, double start, double end).
- void arc (int x, int y, int w, int h, double a1, double a2)

  see fl_arc(int x, int y, int w, int h, double a1, double a2).
- void begin_complex_polygon ()

see fl_begin_complex_polygon().

- void begin_line ()

  see fl_begin_line().
- void begin_loop ()

  see fl_begin_loop().
- void begin_points ()

  see fl_begin_points().
- void begin_polygon ()

  see fl_begin_polygon().
- void circle (double x, double y, double r)

  see fl_circle(double x, double y, double r).
- const char ∗ class_name ()

  Returns the name of the class of this object.
- int clip_box (int x, int y, int w, int h, int &X, int &Y, int &W, int &H)

  see fl_clip_box(int x, int y, int w, int h, int &X, int &Y, int &W, int &H).
- int clocale_printf (const char ∗format,...)

  Shields output PostScript data from modifications of the current locale.
- void color (Fl_Color c)

  see fl_color(Fl_Color c).
- void color (uchar r, uchar g, uchar b)

  see fl_color(uchar r, uchar g, uchar b).
- void curve (double x, double y, double x1, double y1, double x2, double y2, double x3, double y3)

  see fl_curve(double X0, double Y0, double X1, double Y1, double X2, double Y2, double X3, double Y3).
- int descent ()

  see fl_descent().
- void draw (const char ∗s, int nBytes, int x, int y)

  see fl_draw(const char ∗str, int n, int x, int y).
- void draw (int angle, const char ∗str, int n, int x, int y)

  see fl_draw(int angle, const char ∗str, int n, int x, int y).
- void draw (Fl_Pixmap ∗pxm, int XP, int YP, int WP, int HP, int cx, int cy)

  Draws an Fl_Pixmap object to the device.
- void draw (Fl_Bitmap ∗bitmap, int XP, int YP, int WP, int HP, int cx, int cy)

  Draws an Fl_Bitmap object to the device.
- void draw (Fl_RGB_Image ∗rgb, int XP, int YP, int WP, int HP, int cx, int cy)

  Draws an Fl_RGB_Image object to the device.
- void draw_image (const uchar ∗d, int x, int y, int w, int h, int delta=3, int ldelta=0)

  see fl_draw_image(const uchar∗ buf, int X,int Y,int W,int H, int D, int L).
- void draw_image (Fl_Draw_Image_Cb call, void ∗data, int x, int y, int w, int h, int delta=3)

  see fl_draw_image(Fl_Draw_Image_Cb cb, void∗ data, int X,int Y,int W,int H, int D).
- void draw_image_mono (const uchar ∗d, int x, int y, int w, int h, int delta=1, int ld=0)

  see fl_draw_image_mono(const uchar∗ buf, int X,int Y,int W,int H, int D, int L).
- void draw_image_mono (Fl_Draw_Image_Cb call, void ∗data, int x, int y, int w, int h, int delta=1)

  see fl_draw_image_mono(Fl_Draw_Image_Cb cb, void∗ data, int X,int Y,int W,int H, int D).
- int draw_scaled (Fl_Image ∗img, int XP, int YP, int WP, int HP)

  Draws an Fl_Image scaled to width `W` & height `H` with top-left corner at X,Y.
- void end_complex_polygon ()

see fl_end_complex_polygon().

- void end_line ()

  see fl_end_line().
- void end_loop ()

  see fl_end_loop().
- void end_points ()

  see fl_end_points().
- void end_polygon ()

  see fl_end_polygon().
- Fl_PostScript_Graphics_Driver ()

  The constructor.
- void font (int face, int size)

  see fl_font(Fl_Font face, Fl_Fontsize size).
- void gap ()

  see fl_gap().
- int height ()

  see fl_height().
- void line (int x1, int y1, int x2, int y2)

  see fl_line(int x, int y, int x1, int y1).
- void line (int x1, int y1, int x2, int y2, int x3, int y3)

  see fl_line(int x, int y, int x1, int y1, int x2, int y2).
- void line_style (int style, int width=0, char ∗dashes=0)

  see fl_line_style(int style, int width, char∗ dashes).
- void loop (int x0, int y0, int x1, int y1, int x2, int y2)

  see fl_loop(int x0, int y0, int x1, int y1, int x2, int y2).
- void loop (int x0, int y0, int x1, int y1, int x2, int y2, int x3, int y3)

  see fl_loop(int x0, int y0, int x1, int y1, int x2, int y2, int x3, int y3).
- int not_clipped (int x, int y, int w, int h)

  see fl_not_clipped(int x, int y, int w, int h).
- void pie (int x, int y, int w, int h, double a1, double a2)

  see fl_pie(int x, int y, int w, int h, double a1, double a2).
- void point (int x, int y)

  see fl_point(int x, int y).
- void polygon (int x0, int y0, int x1, int y1, int x2, int y2)

  see fl_polygon(int x0, int y0, int x1, int y1, int x2, int y2).
- void polygon (int x0, int y0, int x1, int y1, int x2, int y2, int x3, int y3)

  see fl_polygon(int x0, int y0, int x1, int y1, int x2, int y2, int x3, int y3).
- void pop_clip ()

  see fl_pop_clip().
- void push_clip (int x, int y, int w, int h)

  see fl_push_clip(int x, int y, int w, int h).
- void push_no_clip ()

  see fl_push_no_clip().
- void rect (int x, int y, int w, int h)

  see fl_rect(int x, int y, int w, int h).
- void rectf (int x, int y, int w, int h)

see fl_rectf(int x, int y, int w, int h).

- void rtl_draw (const char ∗s, int n, int x, int y)

  see fl_rtl_draw(const char ∗str, int n, int x, int y).
- void text_extents (const char ∗c, int n, int &dx, int &dy, int &w, int &h)

  see fl_text_extents(const char∗, int n, int& dx, int& dy, int& w, int& h).
- void transformed_vertex (double x, double y)

  see fl_transformed_vertex(double xf, double yf).
- void vertex (double x, double y)

  see fl_vertex(double x, double y).
- double width (const char ∗, int)

  see fl_width(const char ∗str, int n).
- double width (unsigned int u)

  see fl_width(unsigned int n).
- void xyline (int x, int y, int x1)

  see fl_xyline(int x, int y, int x1).
- void xyline (int x, int y, int x1, int y2)

  see fl_xyline(int x, int y, int x1, int y2).
- void xyline (int x, int y, int x1, int y2, int x3)

  see fl_xyline(int x, int y, int x1, int y2, int x3).
- void yxline (int x, int y, int y1)

  see fl_yxline(int x, int y, int y1).
- void yxline (int x, int y, int y1, int x2)

  see fl_yxline(int x, int y, int y1, int x2).
- void yxline (int x, int y, int y1, int x2, int y3)

  see fl_yxline(int x, int y, int y1, int x2, int y3).
- ~Fl_PostScript_Graphics_Driver ()

  The destructor.

Static Public Attributes

- static const char ∗ **class_id** = ”Fl_PostScript_Graphics_Driver”

Additional Inherited Members

31.101.1 Detailed Description

PostScript graphical backend.

PostScript text uses vectorial fonts when using the FLTK standard fonts and the latin alphabet or a few other characters listed in the following table. The latin alphabet means all unicode characters between U+0020 and U+017F, or, in other words, the ASCII, Latin-1 Supplement and Latin Extended-A charts.

Char	Code-point	Name	Char	Code-point	Name	Char	Code-point	Name
ƒ	U+0192	florin		U+201A	quotesinglbase	™	U+2122	trademark
ˆ	U+02C6	circumflex	“	U+201C	quotedblleft		U+2202	partialdiff

ˇ	U+02C7	caron	”	U+201-D	quoted-blright		U+2206	Delta
˘	U+02-D8	breve		U+201E	quoted-blbase		U+2211	summa-tion
	U+02-D9	dotac-cent	†	U+2020	dagger		U+221-A	radical
	U+02D-A	ring	‡	U+2021	dag-gerdbl		U+221E	infinity
	U+02D-B	ogonek	•	U+2022	bullet		U+2260	notequal
˜	U+02D-C	tilde	. . .	U+2026	ellipsis		U+2264	lesse-qual
˝	U+02D-D	hun-garum-laut	‰	U+2030	perthou-sand		U+2265	greaterequal
–	U+2013	endash		U+2039	guils-inglleft		U+25C-A	lozenge
—	U+2014	emdash		U+203-A	guils-inglright		U+F-B01	fi
‘	U+2018	quoteleft	/	U+2044	fraction		U+F-B02	fl
’	U+2019	quo-teright	€	U+20A-C	Euro		U+F8FF	apple (Mac OS only)

All other unicode characters or all other fonts (FL_FREE_FONT and above) are output as a bitmap. FLTK standard fonts are output using the corresponding PostScript standard fonts.

31.101.2 Constructor & Destructor Documentation

Fl_PostScript_Graphics_Driver::∼Fl_PostScript_Graphics_Driver ( )

The destructor.

31.101.3 Member Function Documentation

void Fl_PostScript_Graphics_Driver::arc (double *x*, double *y*, double *r*, double *start*, double *end*) `[virtual]`

see fl_arc(double x, double y, double r, double start, double end).

Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::arc (int *x*, int *y*, int *w*, int *h*, double *a1*, double *a2*) `[virtual]`

see fl_arc(int x, int y, int w, int h, double a1, double a2).

Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::begin_complex_polygon ( ) `[inline]`, `[virtual]`

see fl_begin_complex_polygon().

Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::begin_line ( ) [virtual]

see fl_begin_line().

Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::begin_loop ( ) [virtual]

see fl_begin_loop().

Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::begin_points ( ) [virtual]

see fl_begin_points().

Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::begin_polygon ( ) [virtual]

see fl_begin_polygon().

Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::circle (double *x*, double *y*, double *r*) [virtual]

see fl_circle(double x, double y, double r).

Reimplemented from Fl_Graphics_Driver.

const char∗ Fl_PostScript_Graphics_Driver::class_name ( ) [inline], [virtual]

Returns the name of the class of this object.

Use of the class_name() function is discouraged because it will be removed from future FLTK versions.

The class of an instance of an Fl_Device subclass can be checked with code such as:

```
if ( instance->class_name() == Fl_Printer::class_id ) { ... }
```

Reimplemented from Fl_Graphics_Driver.

int Fl_PostScript_Graphics_Driver::clip_box (int *x*, int *y*, int *w*, int *h*, int & *X*, int & *Y*, int & *W*, int & *H*) [virtual]

see fl_clip_box(int x, int y, int w, int h, int &X, int &Y, int &W, int &H).

Reimplemented from Fl_Graphics_Driver.

int Fl_PostScript_Graphics_Driver::clocale_printf (const char ∗ *format*, ...)

Shields output PostScript data from modifications of the current locale.

It typically avoids PostScript errors caused if the current locale uses comma instead of dot as ”decimal point”.

Parameters

format	directives controlling output PostScript data

Returns

value returned by vfprintf() call

void Fl_PostScript_Graphics_Driver::color (Fl_Color *c*) `[virtual]`

see fl_color(Fl_Color c).

Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::color (uchar *r*, uchar *g*, uchar *b*) `[virtual]`

see fl_color(uchar r, uchar g, uchar b).

Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::curve (double *X0*, double *Y0*, double *X1*, double *Y1*, double *X2*, double *Y2*, double *X3*, double *Y3*) `[virtual]`

see fl_curve(double X0, double Y0, double X1, double Y1, double X2, double Y2, double X3, double Y3).

Reimplemented from Fl_Graphics_Driver.

int Fl_PostScript_Graphics_Driver::descent ( ) `[virtual]`

see fl_descent().

Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::draw (const char ∗ *str*, int *n*, int *x*, int *y*) `[inline]`, `[virtual]`

see fl_draw(const char ∗str, int n, int x, int y).

Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::draw (int *angle*, const char ∗ *str*, int *n*, int *x*, int *y*) `[virtual]`

see fl_draw(int angle, const char ∗str, int n, int x, int y).

Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::draw (Fl_Pixmap ∗ *pxm*, int *XP*, int *YP*, int *WP*, int *HP*, int *cx*, int *cy*) `[virtual]`

Draws an Fl_Pixmap object to the device.

Specifies a bounding box for the image, with the origin (upper left-hand corner) of the image offset by the cx and cy arguments.

Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::draw (Fl_Bitmap ∗ *bm*, int *XP*, int *YP*, int *WP*, int *HP*, int *cx*, int *cy*) `[virtual]`

Draws an Fl_Bitmap object to the device.

Specifies a bounding box for the image, with the origin (upper left-hand corner) of the image offset by the cx and cy arguments.

Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::draw (Fl_RGB_Image ∗ *rgb*, int *XP*, int *YP*, int *WP*, int *HP*, int *cx*, int *cy*) `[virtual]`

Draws an Fl_RGB_Image object to the device.

Specifies a bounding box for the image, with the origin (upper left-hand corner) of the image offset by the cx and cy arguments.

Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::draw_image (const uchar ∗ *buf,* int *X,* int *Y,* int *W,* int *H,* int *D* = *3,* int *L* = *0*) `[virtual]`

see fl_draw_image(const uchar∗ buf, int X,int Y,int W,int H, int D, int L).
Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::draw_image (Fl_Draw_Image_Cb *cb,* void ∗ *data,* int *X,* int *Y,* int *W,* int *H,* int *D* = *3*) `[virtual]`

see fl_draw_image(Fl_Draw_Image_Cb cb, void∗ data, int X,int Y,int W,int H, int D).
Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::draw_image_mono (const uchar ∗ *buf,* int *X,* int *Y,* int *W,* int *H,* int *D* = *1,* int *L* = *0*) `[virtual]`

see fl_draw_image_mono(const uchar∗ buf, int X,int Y,int W,int H, int D, int L).
Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::draw_image_mono (Fl_Draw_Image_Cb *cb,* void ∗ *data,* int *X,* int *Y,* int *W,* int *H,* int *D* = *1*) `[virtual]`

see fl_draw_image_mono(Fl_Draw_Image_Cb cb, void∗ data, int X,int Y,int W,int H, int D).
Reimplemented from Fl_Graphics_Driver.

int Fl_PostScript_Graphics_Driver::draw_scaled (Fl_Image ∗ *img,* int *X,* int *Y,* int *W,* int *H*) `[virtual]`

Draws an Fl_Image scaled to width `W` & height `H` with top-left corner at *X*,Y.

Returns

zero when the graphics driver doesn't implement scaled drawing, non-zero if it does implement it.

Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::end_complex_polygon ( ) `[inline]`, `[virtual]`

see fl_end_complex_polygon().
Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::end_line ( ) `[virtual]`

see fl_end_line().
Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::end_loop ( ) `[virtual]`

see fl_end_loop().
Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::end_points ( ) `[virtual]`

see fl_end_points().
Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::end_polygon ( ) `[virtual]`

see fl_end_polygon().

Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::font (int *face*, int *fsize*) `[virtual]`

see fl_font(Fl_Font face, Fl_Fontsize size).

Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::gap ( ) `[inline]`, `[virtual]`

see fl_gap().

Reimplemented from Fl_Graphics_Driver.

int Fl_PostScript_Graphics_Driver::height ( ) `[virtual]`

see fl_height().

Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::line (int *x*, int *y*, int *x1*, int *y1*) `[virtual]`

see fl_line(int x, int y, int x1, int y1).

Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::line (int *x*, int *y*, int *x1*, int *y1*, int *x2*, int *y2*) `[virtual]`

see fl_line(int x, int y, int x1, int y1, int x2, int y2).

Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::line_style (int *style*, int *width* = `0`, char ∗ *dashes* = `0`) `[virtual]`

see fl_line_style(int style, int width, char∗ dashes).

Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::loop (int *x0*, int *y0*, int *x1*, int *y1*, int *x2*, int *y2*) `[virtual]`

see fl_loop(int x0, int y0, int x1, int y1, int x2, int y2).

Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::loop (int *x0*, int *y0*, int *x1*, int *y1*, int *x2*, int *y2*, int *x3*, int *y3*) `[virtual]`

see fl_loop(int x0, int y0, int x1, int y1, int x2, int y2, int x3, int y3).

Reimplemented from Fl_Graphics_Driver.

int Fl_PostScript_Graphics_Driver::not_clipped (int *x*, int *y*, int *w*, int *h*) `[virtual]`

see fl_not_clipped(int x, int y, int w, int h).

Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::pie (int *x*, int *y*, int *w*, int *h*, double *a1*, double *a2*) `[virtual]`

see fl_pie(int x, int y, int w, int h, double a1, double a2).

Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::point (int *x*, int *y*) `[virtual]`

see fl_point(int x, int y).

Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::polygon (int *x0*, int *y0*, int *x1*, int *y1*, int *x2*, int *y2*) `[virtual]`

see fl_polygon(int x0, int y0, int x1, int y1, int x2, int y2).

Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::polygon (int *x0*, int *y0*, int *x1*, int *y1*, int *x2*, int *y2*, int *x3*, int *y3*) `[virtual]`

see fl_polygon(int x0, int y0, int x1, int y1, int x2, int y2, int x3, int y3).

Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::pop_clip ( ) `[virtual]`

see fl_pop_clip().

Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::push_clip (int *x*, int *y*, int *w*, int *h*) `[virtual]`

see fl_push_clip(int x, int y, int w, int h).

Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::push_no_clip ( ) `[virtual]`

see fl_push_no_clip().

Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::rect (int *x*, int *y*, int *w*, int *h*) `[virtual]`

see fl_rect(int x, int y, int w, int h).

Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::rectf (int *x*, int *y*, int *w*, int *h*) `[virtual]`

see fl_rectf(int x, int y, int w, int h).

Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::rtl_draw (const char ∗ *str*, int *n*, int *x*, int *y*) `[virtual]`

see fl_rtl_draw(const char ∗str, int n, int x, int y).

Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::text_extents (const char ∗ *t*, int *n*, int & *dx*, int & *dy*, int & *w*, int & *h*) `[virtual]`

see fl_text_extents(const char∗, int n, int& dx, int& dy, int& w, int& h).
Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::transformed_vertex (double *xf*, double *yf*) `[virtual]`

see fl_transformed_vertex(double xf, double yf).
Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::vertex (double *x*, double *y*) `[virtual]`

see fl_vertex(double x, double y).
Reimplemented from Fl_Graphics_Driver.

double Fl_PostScript_Graphics_Driver::width (const char ∗ *str*, int *n*) `[virtual]`

see fl_width(const char ∗str, int n).
Reimplemented from Fl_Graphics_Driver.

double Fl_PostScript_Graphics_Driver::width (unsigned int *c*) `[virtual]`

see fl_width(unsigned int n).
Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::xyline (int *x*, int *y*, int *x1*) `[virtual]`

see fl_xyline(int x, int y, int x1).
Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::xyline (int *x*, int *y*, int *x1*, int *y2*) `[virtual]`

see fl_xyline(int x, int y, int x1, int y2).
Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::xyline (int *x*, int *y*, int *x1*, int *y2*, int *x3*) `[virtual]`

see fl_xyline(int x, int y, int x1, int y2, int x3).
Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::yxline (int *x*, int *y*, int *y1*) `[virtual]`

see fl_yxline(int x, int y, int y1).
Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::yxline (int *x*, int *y*, int *y1*, int *x2*) `[virtual]`

see fl_yxline(int x, int y, int y1, int x2).
Reimplemented from Fl_Graphics_Driver.

void Fl_PostScript_Graphics_Driver::yxline (int *x*, int *y*, int *y1*, int *x2*, int *y3*) `[virtual]`

see fl_yxline(int x, int y, int y1, int x2, int y3).

Reimplemented from Fl_Graphics_Driver.

The documentation for this class was generated from the following files:

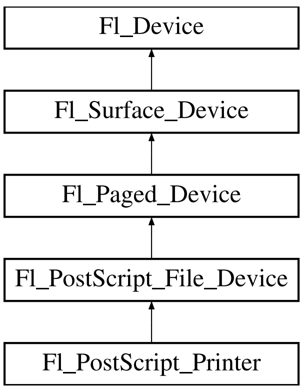
- Fl_PostScript.H
- Fl_PostScript.cxx

31.102 Fl_PostScript_Printer Class Reference

Print support under Unix/Linux.

`#include <Fl_Printer.H>`

Inheritance diagram for Fl_PostScript_Printer:



Public Member Functions

- const char ∗ class_name ()

  Returns the name of the class of this object.
- int start_job (int pages, int ∗firstpage=NULL, int ∗lastpage=NULL)

  Starts a print job.

Static Public Attributes

- static const char ∗ **class_id** = Fl_Printer::class_id

Protected Member Functions

- Fl_PostScript_Printer (void)

  The constructor.

Friends

- class **Fl_Printer**

Additional Inherited Members

31.102.1 Detailed Description

Print support under Unix/Linux.

Class Fl_PostScript_Printer is implemented only on the Unix/Linux platform. It has no public constructor. Use Fl_Printer instead that is cross-platform and has the same API.

31.102.2 Member Function Documentation

const char∗ Fl_PostScript_Printer::class_name ( ) `[inline]`, `[virtual]`

Returns the name of the class of this object.

Use of the class_name() function is discouraged because it will be removed from future FLTK versions.

The class of an instance of an Fl_Device subclass can be checked with code such as:

```
if ( instance->class_name() == Fl_Printer::class_id ) { ... }
```

Reimplemented from Fl_PostScript_File_Device.

int Fl_PostScript_Printer::start_job (int *pages*, int ∗ *firstpage* = `NULL`, int ∗ *lastpage* = `NULL`) `[virtual]`

Starts a print job.

Reimplemented from Fl_PostScript_File_Device.

The documentation for this class was generated from the following files:

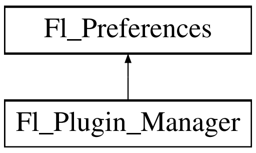
- Fl_Printer.H
- Fl_PostScript.cxx
- Fl_Printer.cxx

31.103 Fl_Preferences Class Reference

Fl_Preferences provides methods to store user settings between application starts.

`#include <Fl_Preferences.H>`

Inheritance diagram for Fl_Preferences:



Classes

- struct Entry
- class Name

  'Name' provides a simple method to create numerical or more complex procedural names for entries and groups on the fly.
- class Node
- class RootNode

Public Types

- typedef void ∗ ID

  Every Fl_Preferences-Group has a uniqe ID.
- enum Root { SYSTEM =0, USER }

  Define the scope of the preferences.

Public Member Functions

- char clear ()

  Delete all groups and all entries.
- char deleteAllEntries ()

  Delete all entries.
- char deleteAllGroups ()

  Delete all groups.
- char deleteEntry (const char ∗entry)

  Deletes a single name/value pair.
- char deleteGroup (const char ∗group)

  Deletes a group.
- int entries ()

  Returns the number of entries (name/value pairs) in a group.
- const char ∗ entry (int index)

  Returns the name of an entry.
- char entryExists (const char ∗key)

  Returns non-zero if an entry with this name exists.
- Fl_Preferences (Root root, const char ∗vendor, const char ∗application)

  The constructor creates a group that manages name/value pairs and child groups.
- Fl_Preferences (const char ∗path, const char ∗vendor, const char ∗application)

  Use this constructor to create or read a preferences file at an arbitrary position in the file system.
- Fl_Preferences (Fl_Preferences &parent, const char ∗group)

  Generate or read a new group of entries within another group.
- Fl_Preferences (Fl_Preferences ∗parent, const char ∗group)

  Create or access a group of preferences using a name.
- Fl_Preferences (Fl_Preferences &parent, int groupIndex)

  Open a child group using a given index.
- Fl_Preferences (Fl_Preferences ∗parent, int groupIndex)
- Fl_Preferences (const Fl_Preferences &)

  Create another reference to a Preferences group.
- Fl_Preferences (ID id)

  Create a new dataset access point using a dataset ID.
- void flush ()

  Writes all preferences to disk.
- char get (const char ∗entry, int &value, int defaultValue)

  Reads an entry from the group.
- char get (const char ∗entry, float &value, float defaultValue)

  Reads an entry from the group.
- char get (const char ∗entry, double &value, double defaultValue)

  Reads an entry from the group.

- char get (const char ∗entry, char ∗&value, const char ∗defaultValue)

  Reads an entry from the group.
- char get (const char ∗entry, char ∗value, const char ∗defaultValue, int maxSize)

  Reads an entry from the group.
- char get (const char ∗entry, void ∗&value, const void ∗defaultValue, int defaultSize)

  Reads an entry from the group.
- char get (const char ∗entry, void ∗value, const void ∗defaultValue, int defaultSize, int maxSize)

  Reads an entry from the group.
- char getUserdataPath (char ∗path, int pathlen)

  Creates a path that is related to the preferences file and that is usable for additional application data.
- const char ∗ group (int num_group)

  Returns the name of the Nth (`num_group`) group.
- char groupExists (const char ∗key)

  Returns non-zero if a group with this name exists.
- int groups ()

  Returns the number of groups that are contained within a group.
- ID id ()

  Return an ID that can later be reused to open more references to this dataset.
- const char ∗ name ()

  Return the name of this entry.
- const char ∗ path ()

  Return the full path to this entry.
- char set (const char ∗entry, int value)

  Sets an entry (name/value pair).
- char set (const char ∗entry, float value)

  Sets an entry (name/value pair).
- char set (const char ∗entry, float value, int precision)

  Sets an entry (name/value pair).
- char set (const char ∗entry, double value)

  Sets an entry (name/value pair).
- char set (const char ∗entry, double value, int precision)

  Sets an entry (name/value pair).
- char set (const char ∗entry, const char ∗value)

  Sets an entry (name/value pair).
- char set (const char ∗entry, const void ∗value, int size)

  Sets an entry (name/value pair).
- int size (const char ∗entry)

  Returns the size of the value part of an entry.
- virtual ∼Fl_Preferences ()

  The destructor removes allocated resources.

Static Public Member Functions

- static const char ∗ newUUID ()

  Returns a UUID as generated by the system.
- static char remove (ID id_)

  Remove the group with this ID from a database.

Protected Attributes

- Node ∗ **node**
- RootNode ∗ **rootNode**

Friends

- class **Node**
- class **RootNode**

31.103.1 Detailed Description

Fl_Preferences provides methods to store user settings between application starts.

It is similar to the Registry on WIN32 and Preferences on MacOS, and provides a simple configuration mechanism for UNIX.

Fl_Preferences uses a hierarchy to store data. It bundles similar data into groups and manages entries into those groups as name/value pairs.

Preferences are stored in text files that can be edited manually. The file format is easy to read and relatively forgiving. Preferences files are the same on all platforms. User comments in preference files are preserved. Filenames are unique for each application by using a vendor/application naming scheme. The user must provide default values for all entries to ensure proper operation should preferences be corrupted or not yet exist.

Entries can be of any length. However, the size of each preferences file should be kept small for performance reasons. One application can have multiple preferences files. Extensive binary data however should be stored in separate files: see getUserdataPath().

Note

> Starting with FLTK 1.3, preference databases are expected to be in UTF-8 encoding. Previous databases were stored in the current character set or code page which renders them incompatible for text entries using international characters.

31.103.2 Member Typedef Documentation

typedef void∗ Fl_Preferences::ID

Every Fl_Preferences-Group has a uniqe ID.

ID's can be retrieved from an Fl_Preferences-Group and can then be used to create more Fl_Preference references to the same data set, as long as the database remains open.

31.103.3 Member Enumeration Documentation

enum Fl_Preferences::Root

Define the scope of the preferences.

Enumerator

SYSTEM Preferences are used system-wide.

USER Preferences apply only to the current user.

31.103.4 Constructor & Destructor Documentation

Fl_Preferences::Fl_Preferences (Root *root*, const char ∗ *vendor*, const char ∗ *application*)

The constructor creates a group that manages name/value pairs and child groups.

Groups are ready for reading and writing at any time. The root argument is either Fl_Preferences::USER or Fl_Preferences::SYSTEM.

This constructor creates the *base* instance for all following entries and reads existing databases into memory. The vendor argument is a unique text string identifying the development team or vendor of an application. A domain name or an EMail address are great unique names, e.g. "researchATmatthiasm.-com" or "fltk.org". The application argument can be the working title or final name of your application. Both vendor and application must be valid relative UNIX pathnames and may contain '/'s to create deeper file structures.

A set of Preferences marked "run-time" exists exactly one per application and only as long as the application runs. It can be used as a database for volatile information. FLTK uses it to register plugins at run-time.

Parameters

`in`	*root*	can be `USER` or `SYSTEM` for user specific or system wide preferences
`in`	*vendor*	unique text describing the company or author of this file
`in`	*application*	unique text describing the application

Fl_Preferences::Fl_Preferences (const char ∗ *path*, const char ∗ *vendor*, const char ∗ *application*)

Use this constructor to create or read a preferences file at an arbitrary position in the file system.

The file name is generated in the form `path/application.prefs`. If `application` is `NULL`, `path` must contain the full file name.

Parameters

`in`	*path*	path to the directory that contains the preferences file
`in`	*vendor*	unique text describing the company or author of this file
`in`	*application*	unique text describing the application

Fl_Preferences::Fl_Preferences (Fl_Preferences & *parent*, const char ∗ *group*)

Generate or read a new group of entries within another group.

Use the `group` argument to name the group that you would like to access. `Group` can also contain a path to a group further down the hierarchy by separating group names with a forward slash '/'.

Parameters

`in`	*parent*	reference object for the new group
`in`	*group*	name of the group to access (may contain '/'s)

Fl_Preferences::Fl_Preferences (Fl_Preferences ∗ *parent*, const char ∗ *group*)

Create or access a group of preferences using a name.

Parameters

`in`	*parent*	the parameter parent is a pointer to the parent group. `Parent` may be `NULL`. It then refers to an application internal database which exists only once, and remains in RAM only until the application quits. This database is used to manage plugins and other data indexes by strings.
`in`	*group*	a group name that is used as a key into the database

See Also

Fl_Preferences(Fl_Preferences&, const char ∗group)

Fl_Preferences::Fl_Preferences (Fl_Preferences & *parent,* int *groupIndex*)

Open a child group using a given index.

Use the `groupIndex` argument to find the group that you would like to access. If the given index is invalid (negative or too high), a new group is created with a UUID as a name.

The index needs to be fixed. It is currently backward. Index 0 points to the last member in the 'list' of preferences.

Parameters

`in`	*parent*	reference object for the new group
`in`	*groupIndex*	zero based index into child groups

Fl_Preferences::Fl_Preferences (Fl_Preferences ∗ *parent,* int *groupIndex*)

See Also

Fl_Preferences(Fl_Preferences&, int groupIndex)

Fl_Preferences::Fl_Preferences (Fl_Preferences::ID *id*)

Create a new dataset access point using a dataset ID.

ID's are a great way to remember shortcuts to database entries that are deeply nested in a preferences database, as long as the database root is not deleted. An ID can be retrieved from any Fl_Preferences dataset, and can then be used to create multiple new references to the same dataset.

ID's can be very helpful when put into the `user_data()` field of widget callbacks.

Fl_Preferences::∼Fl_Preferences ( ) `[virtual]`

The destructor removes allocated resources.

When used on the *base* preferences group, the destructor flushes all changes to the preferences file and deletes all internal databases.

The destructor does not remove any data from the database. It merely deletes your reference to the database.

31.103.5 Member Function Documentation

char Fl_Preferences::deleteEntry (const char ∗ *key*)

Deletes a single name/value pair.

This function removes the entry `key` from the database.

Parameters

`in`	*key*	name of entry to delete

Returns

0 if deleting the entry failed

char Fl_Preferences::deleteGroup (const char ∗ *group*)

Deletes a group.

Removes a group and all keys and groups within that group from the database.

Parameters

`in`	*group*	name of the group to delete

Returns

0 if call failed

int Fl_Preferences::entries ( )

Returns the number of entries (name/value pairs) in a group.

Returns

number of entries

const char ∗ Fl_Preferences::entry (int *index*)

Returns the name of an entry.
There is no guaranteed order of entry names. The index must be within the range given by entries().
Parameters

`in`	*index*	number indexing the requested entry

Returns

pointer to value cstring

char Fl_Preferences::entryExists (const char ∗ *key*)

Returns non-zero if an entry with this name exists.
Parameters

`in`	*key*	name of entry that is searched for

Returns

0 if entry was not found

void Fl_Preferences::flush ( )

Writes all preferences to disk.
This function works only with the base preferences group. This function is rarely used as deleting the base preferences flushes automatically.

char Fl_Preferences::get (const char ∗ *key,* int & *value,* int *defaultValue*)

Reads an entry from the group.
A default value must be supplied. The return value indicates if the value was available (non-zero) or the default was used (0).

Parameters

in	*key*	name of entry
out	*value*	returned from preferences or default value if none was set
in	*defaultValue*	default value to be used if no preference was set

Returns

0 if the default value was used

char Fl_Preferences::get (const char ∗ *key*, float & *value*, float *defaultValue*)

Reads an entry from the group.

A default value must be supplied. The return value indicates if the value was available (non-zero) or the default was used (0).

Parameters

in	*key*	name of entry
out	*value*	returned from preferences or default value if none was set
in	*defaultValue*	default value to be used if no preference was set

Returns

0 if the default value was used

char Fl_Preferences::get (const char ∗ *key*, double & *value*, double *defaultValue*)

Reads an entry from the group.

A default value must be supplied. The return value indicates if the value was available (non-zero) or the default was used (0).

Parameters

in	*key*	name of entry
out	*value*	returned from preferences or default value if none was set
in	*defaultValue*	default value to be used if no preference was set

Returns

0 if the default value was used

char Fl_Preferences::get (const char ∗ *key*, char ∗& *text*, const char ∗ *defaultValue*)

Reads an entry from the group.

A default value must be supplied. The return value indicates if the value was available (non-zero) or the default was used (0). get() allocates memory of sufficient size to hold the value. The buffer must be free'd by the developer using 'free(value)'.

Parameters

in	*key*	name of entry
out	*text*	returned from preferences or default value if none was set

in	*defaultValue*	default value to be used if no preference was set

Returns

0 if the default value was used

char Fl_Preferences::get (const char ∗ *key,* char ∗ *text,* const char ∗ *defaultValue,* int *maxSize*)

Reads an entry from the group.

A default value must be supplied. The return value indicates if the value was available (non-zero) or the default was used (0). 'maxSize' is the maximum length of text that will be read. The text buffer must allow for one additional byte for a trailing zero.

Parameters

in	*key*	name of entry
out	*text*	returned from preferences or default value if none was set
in	*defaultValue*	default value to be used if no preference was set
in	*maxSize*	maximum length of value plus one byte for a trailing zero

Returns

0 if the default value was used

char Fl_Preferences::get (const char ∗ *key,* void ∗& *data,* const void ∗ *defaultValue,* int *defaultSize*)

Reads an entry from the group.

A default value must be supplied. The return value indicates if the value was available (non-zero) or the default was used (0). get() allocates memory of sufficient size to hold the value. The buffer must be free'd by the developer using 'free(value)'.

Parameters

in	*key*	name of entry
out	*data*	returned from preferences or default value if none was set
in	*defaultValue*	default value to be used if no preference was set
in	*defaultSize*	size of default value array

Returns

0 if the default value was used

char Fl_Preferences::get (const char ∗ *key,* void ∗ *data,* const void ∗ *defaultValue,* int *defaultSize,* int *maxSize*)

Reads an entry from the group.

A default value must be supplied. The return value indicates if the value was available (non-zero) or the default was used (0). 'maxSize' is the maximum length of text that will be read.

Parameters

in	*key*	name of entry
out	*data*	value returned from preferences or default value if none was set
in	*defaultValue*	default value to be used if no preference was set
in	*defaultSize*	size of default value array
in	*maxSize*	maximum length of value

Returns

0 if the default value was used

Todo maxSize should receive the number of bytes that were read.

char Fl_Preferences::getUserdataPath (char ∗ *path*, int *pathlen*)

Creates a path that is related to the preferences file and that is usable for additional application data.

This function creates a directory that is named after the preferences database without the .prefs extension and located in the same directory. It then fills the given buffer with the complete path name.

Example:

```
Fl_Preferences prefs( USER, "matthiasm.com", "test" );
char path[FL_PATH_MAX];
prefs.getUserdataPath( path );
```

..creates the preferences database in (MS Windows):

```
c:/Documents and Settings/matt/Application Data/matthiasm.com/test.prefs
```

..and returns the userdata path:

```
c:/Documents and Settings/matt/Application Data/matthiasm.com/test/
```

Parameters

out	*path*	buffer for user data path
in	*pathlen*	size of path buffer (should be at least `FL_PATH_MAX`)

Returns

0 if path was not created or pathname can't fit into buffer

const char ∗ Fl_Preferences::group (int *num_group*)

Returns the name of the Nth (`num_group`) group.

There is no guaranteed order of group names. The index must be within the range given by groups().

Parameters

in	*num_group*	number indexing the requested group

Returns

'C' string pointer to the group name

char Fl_Preferences::groupExists (const char ∗ *key*)

Returns non-zero if a group with this name exists.

Group names are relative to the Preferences node and can contain a path. "." describes the current node, "./" describes the topmost node. By preceding a groupname with a "./", its path becomes relative to the topmost node.

Parameters

in	*key*	name of group that is searched for

Returns

0 if no group by that name was found

int Fl_Preferences::groups ( )

Returns the number of groups that are contained within a group.

Returns

0 for no groups at all

const char * Fl_Preferences::newUUID ( ) `[static]`

Returns a UUID as generated by the system.

A UUID is a "universally unique identifier" which is commonly used in configuration files to create identities. A UUID in ASCII looks like this: `937C4900-51AA-4C11-8DD3-7AB59944F03E`. It has always 36 bytes plus a trailing zero.

Returns

a pointer to a static buffer containing the new UUID in ASCII format. The buffer is overwritten during every call to this function!

char Fl_Preferences::set (const char * *key,* int *value*)

Sets an entry (name/value pair).

The return value indicates if there was a problem storing the data in memory. However it does not reflect if the value was actually stored in the preferences file.

Parameters

in	*key*	name of entry
in	*value*	set this entry to `value`

Returns

0 if setting the value failed

char Fl_Preferences::set (const char * *key,* float *value*)

Sets an entry (name/value pair).

The return value indicates if there was a problem storing the data in memory. However it does not reflect if the value was actually stored in the preferences file.

Parameters

in	*key*	name of entry

`in`	*value*	set this entry to `value`

Returns

0 if setting the value failed

char Fl_Preferences::set (const char * *key*, float *value*, int *precision*)

Sets an entry (name/value pair).

The return value indicates if there was a problem storing the data in memory. However it does not reflect if the value was actually stored in the preferences file.

Parameters

`in`	*key*	name of entry
`in`	*value*	set this entry to `value`
`in`	*precision*	number of decimal digits to represent value

Returns

0 if setting the value failed

char Fl_Preferences::set (const char * *key*, double *value*)

Sets an entry (name/value pair).

The return value indicates if there was a problem storing the data in memory. However it does not reflect if the value was actually stored in the preferences file.

Parameters

`in`	*key*	name of entry
`in`	*value*	set this entry to `value`

Returns

0 if setting the value failed

char Fl_Preferences::set (const char * *key*, double *value*, int *precision*)

Sets an entry (name/value pair).

The return value indicates if there was a problem storing the data in memory. However it does not reflect if the value was actually stored in the preferences file.

Parameters

`in`	*key*	name of entry
`in`	*value*	set this entry to `value`
`in`	*precision*	number of decimal digits to represent value

Returns

0 if setting the value failed

char Fl_Preferences::set (const char * *key*, const char * *text*)

Sets an entry (name/value pair).

The return value indicates if there was a problem storing the data in memory. However it does not reflect if the value was actually stored in the preferences file.

Parameters

in	*key*	name of entry
in	*text*	set this entry to `value`

Returns

0 if setting the value failed

char Fl_Preferences::set (const char ∗ ***key,* const void** ∗ ***data,* int *dsize*)**

Sets an entry (name/value pair).

The return value indicates if there was a problem storing the data in memory. However it does not reflect if the value was actually stored in the preferences file.

Parameters

in	*key*	name of entry
in	*data*	set this entry to `value`
in	*dsize*	size of data array

Returns

0 if setting the value failed

int Fl_Preferences::size (const char ∗ ***key*)**

Returns the size of the value part of an entry.

Parameters

in	*key*	name of entry

Returns

size of value

The documentation for this class was generated from the following files:

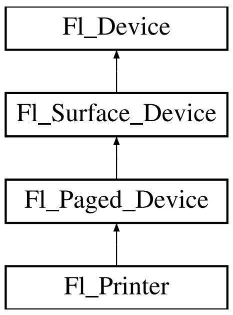
- Fl_Preferences.H
- Fl_Preferences.cxx

31.104 Fl_Printer Class Reference

OS-independent print support.

`#include <Fl_Printer.H>`

Inheritance diagram for Fl_Printer:



Public Member Functions

- const char * class_name ()

  Returns the name of the class of this object.
- Fl_Graphics_Driver * **driver** (void)
- void end_job (void)

  To be called at the end of a print job.
- int end_page (void)

  To be called at the end of each page.
- Fl_Printer (void)

  The constructor.
- void margins (int *left, int *top, int *right, int *bottom)

  Computes the dimensions of margins that lie between the printable page area and the full page.
- void origin (int *x, int *y)

  Computes the page coordinates of the current origin of graphics functions.
- void origin (int x, int y)

  Sets the position in page coordinates of the origin of graphics functions.
- void print_widget (Fl_Widget *widget, int delta_x=0, int delta_y=0)

  Draws the widget on the printed page.
- void print_window_part (Fl_Window *win, int x, int y, int w, int h, int delta_x=0, int delta_y=0)

  Prints a rectangular part of an on-screen window.
- int printable_rect (int *w, int *h)

  Computes the width and height of the printable area of the page.
- void rotate (float angle)

  Rotates the graphics operations relatively to paper.
- void scale (float scale_x, float scale_y=0.)

  Changes the scaling of page coordinates.
- void set_current (void)

  Make this surface the current drawing surface.
- int start_job (int pagecount, int *frompage=NULL, int *topage=NULL)

  Starts a print job.
- int start_page (void)

  Starts a new printed page.
- void translate (int x, int y)

  Translates the current graphics origin accounting for the current rotation.
- void untranslate (void)

  Undoes the effect of a previous translate() call.
- ~Fl_Printer (void)

  The destructor.

Static Public Attributes

- static const char * **class_id** = ”Fl_Printer”

These attributes are effective under the Xlib platform only.

- static const char * dialog_title = ”Print”

  [this text may be customized at run-time]
- static const char * dialog_printer = ”Printer:”

[this text may be customized at run-time]

- static const char ∗ dialog_range = ”Print Range”

  [this text may be customized at run-time]
- static const char ∗ dialog_copies = ”Copies”

  [this text may be customized at run-time]
- static const char ∗ dialog_all = ”All”

  [this text may be customized at run-time]
- static const char ∗ dialog_pages = ”Pages”

  [this text may be customized at run-time]
- static const char ∗ dialog_from = ”From:”

  [this text may be customized at run-time]
- static const char ∗ dialog_to = ”To:”

  [this text may be customized at run-time]
- static const char ∗ dialog_properties = ”Properties...”

  [this text may be customized at run-time]
- static const char ∗ dialog_copyNo = ”# Copies:”

  [this text may be customized at run-time]
- static const char ∗ dialog_print_button = ”Print”

  [this text may be customized at run-time]
- static const char ∗ dialog_cancel_button = ”Cancel”

  [this text may be customized at run-time]
- static const char ∗ dialog_print_to_file = ”Print To File”

  [this text may be customized at run-time]
- static const char ∗ property_title = ”Printer Properties”

  [this text may be customized at run-time]
- static const char ∗ property_pagesize = ”Page Size:”

  [this text may be customized at run-time]
- static const char ∗ property_mode = ”Output Mode:”

  [this text may be customized at run-time]
- static const char ∗ property_use = ”Use”

  [this text may be customized at run-time]
- static const char ∗ property_save = ”Save”

  [this text may be customized at run-time]
- static const char ∗ property_cancel = ”Cancel”

  [this text may be customized at run-time]

Additional Inherited Members

31.104.1 Detailed Description

OS-independent print support.

Fl_Printer allows to use all drawing, color, text, image, and clip FLTK functions, and to have them operate on printed page(s). There are two main, non exclusive, ways to use it.

- Print any widget (standard, custom, Fl_Window, Fl_Gl_Window) as it appears on screen, with optional translation, scaling and rotation. This is done by calling print_widget(), print_window() or print_window_part().

- Use a series of FLTK graphics commands (e.g., font, text, lines, colors, clip, image) to compose a page appropriately shaped for printing.

In both cases, begin by start_job(), start_page(), printable_rect() and origin() calls and finish by end_page() and end_job() calls.

Example of use: print a widget centered in a page

```
#include <FL/Fl_Printer.H>
#include <FL/fl_draw.H>
int width, height;
Fl_Widget *widget = ...  // a widget we want printed
Fl_Printer *printer = new Fl_Printer();
if (printer->start_job(1) == 0) {
 printer->start_page();
 printer->printable_rect(&width, &height);
 fl_color(FL_BLACK);
 fl_line_style(FL_SOLID, 2);
 fl_rect(0, 0, width, height);
 fl_font(FL_COURIER, 12);
 time_t now; time(&now); fl_draw(ctime(&now), 0, fl_height());
 printer->origin(width/2, height/2);
 printer->print_widget(widget, -widget->w()/2, -widget->h()/2);
 printer->end_page();
 printer->end_job();
}
delete printer;
```

Platform specifics

- Unix/Linux platforms: Unless it has been previously changed, the default paper size is A4. To change that, press the "Properties" button of the "Print" dialog window opened by an Fl_Printer::start_job() call. This opens a "Printer Properties" window where it's possible to select the adequate paper size. Finally press the "Save" button therein to assign the chosen paper size to the chosen printer for this and all further print operations.

  Class Fl_RGB_Image prints but loses its transparency if it has one. See class Fl_PostScript_Graphics-_Driver for a description of how UTF-8 strings appear in print. Use the static public attributes of this class to set the print dialog to other languages than English. For example, the "Printer:" dialog item Fl_Printer::dialog_printer can be set to French with:

  ```
  Fl_Printer::dialog_printer = "Imprimante:";
  ```

  before creation of the Fl_Printer object. Use Fl_PostScript_File_Device::file_chooser_title to customize the title of the file chooser dialog that opens when using the "Print To File" option of the print dialog.
- MSWindows platform: Transparent Fl_RGB_Image 's don't print with exact transparency on most printers. Fl_RGB_Image 's don't rotate() well. A workaround is to use the print_window_part() call.
- Mac OS X platform: all graphics requests print as on display.

31.104.2 Member Function Documentation

const char* Fl_Printer::class_name ( ) [inline], [virtual]

Returns the name of the class of this object.

Use of the class_name() function is discouraged because it will be removed from future FLTK versions. The class of an instance of an Fl_Device subclass can be checked with code such as:

```
if ( instance->class_name() == Fl_Printer::class_id ) { ... }
```

Reimplemented from Fl_Paged_Device.

int Fl_Printer::end_page (void) [virtual]

To be called at the end of each page.

Returns

0 if OK, non-zero if any error.

Reimplemented from Fl_Paged_Device.

void Fl_Printer::margins (int ∗ *left*, int ∗ *top*, int ∗ *right*, int ∗ *bottom*) `[virtual]`

Computes the dimensions of margins that lie between the printable page area and the full page.

Values are in the same unit as that used by FLTK drawing functions. They are changed by scale() calls.

Parameters

`out`	*left*	If non-null, ∗left is set to the left margin size.
`out`	*top*	If non-null, ∗top is set to the top margin size.
`out`	*right*	If non-null, ∗right is set to the right margin size.
`out`	*bottom*	If non-null, ∗bottom is set to the bottom margin size.

Reimplemented from Fl_Paged_Device.

void Fl_Printer::origin (int ∗ *x*, int ∗ *y*) `[virtual]`

Computes the page coordinates of the current origin of graphics functions.

Parameters

`out`	*x*	If non-null, ∗x is set to the horizontal page offset of graphics origin.
`out`	*y*	Same as above, vertically.

Reimplemented from Fl_Paged_Device.

void Fl_Printer::origin (int *x*, int *y*) `[virtual]`

Sets the position in page coordinates of the origin of graphics functions.

Arguments should be expressed relatively to the result of a previous printable_rect() call. That is, `printable_rect(&w, &h); origin(w/2, 0);` sets the graphics origin at the top center of the page printable area. Origin() calls are not affected by rotate() calls. Successive origin() calls don't combine their effects.

Parameters

`in`	*x*	Horizontal position in page coordinates of the desired origin of graphics functions.
`in`	*y*	Same as above, vertically.

Reimplemented from Fl_Paged_Device.

void Fl_Printer::print_widget (Fl_Widget ∗ *widget*, int *delta_x* = 0, int *delta_y* = 0) `[virtual]`

Draws the widget on the printed page.

The widget's position on the printed page is determined by the last call to origin() and by the optional delta_x and delta_y arguments. Its dimensions are in points unless there was a previous call to scale().

Parameters

`in`	*widget*	Any FLTK widget (e.g., standard, custom, window).
`in`	*delta_x*	Optional horizontal offset for positioning the widget relatively to the current origin of graphics functions.
`in`	*delta_y*	Same as above, vertically.

Reimplemented from Fl_Paged_Device.

void Fl_Printer::print_window_part (Fl_Window ∗ *win*, int *x*, int *y*, int *w*, int *h*, int *delta_x* = 0, int *delta_y* = 0) `[virtual]`

Prints a rectangular part of an on-screen window.

Parameters

win	The window from where to capture.
x	The rectangle left
y	The rectangle top
w	The rectangle width
h	The rectangle height
delta_x	Optional horizontal offset from current graphics origin where to print the captured rectangle.
delta_y	As above, vertically.

Reimplemented from Fl_Paged_Device.

int Fl_Printer::printable_rect (int ∗ *w*, int ∗ *h*) `[virtual]`

Computes the width and height of the printable area of the page.

Values are in the same unit as that used by FLTK drawing functions, are unchanged by calls to origin(), but are changed by scale() calls. Values account for the user-selected paper type and print orientation.

Returns

0 if OK, non-zero if any error

Reimplemented from Fl_Paged_Device.

void Fl_Printer::rotate (float *angle*) `[virtual]`

Rotates the graphics operations relatively to paper.

The rotation is centered on the current graphics origin. Successive rotate() calls don't combine their effects.

Parameters

angle	Rotation angle in counter-clockwise degrees.

Reimplemented from Fl_Paged_Device.

void Fl_Printer::scale (float *scale_x*, float *scale_y* = `0.`) `[virtual]`

Changes the scaling of page coordinates.

This function also resets the origin of graphics functions at top left of printable page area. After a scale() call, do a printable_rect() call to get the new dimensions of the printable page area. Successive scale() calls don't combine their effects.

Parameters

scale_x	Horizontal dimensions of plot are multiplied by this quantity.
scale_y	Same as above, vertically. The value 0. is equivalent to setting `scale_y` = `scale_x`. Thus, scale(factor); is equivalent to scale(factor, factor);

Reimplemented from Fl_Paged_Device.

void Fl_Printer::set_current (void) `[virtual]`

Make this surface the current drawing surface.

This surface will receive all future graphics requests.

Reimplemented from Fl_Surface_Device.

int Fl_Printer::start_job (int *pagecount*, int ∗ *frompage* = `NULL`, int ∗ *topage* = `NULL`) `[virtual]`

Starts a print job.

Opens a platform-specific dialog window allowing the user to set several options including the desired printer and the page orientation. Optionally, the user can also select a range of pages to be printed. This range is returned to the caller that is in charge of sending only these pages for printing.

Parameters

`in`	*pagecount*	the total number of pages of the job (or 0 if you don't know the number of pages)
`out`	*frompage*	if non-null, ∗frompage is set to the first page the user wants printed
`out`	*topage*	if non-null, ∗topage is set to the last page the user wants printed

Returns

0 if OK, non-zero if any error occurred or if the user cancelled the print request.

Reimplemented from Fl_Paged_Device.

int Fl_Printer::start_page (void) `[virtual]`

Starts a new printed page.

The page coordinates are initially in points, i.e., 1/72 inch, and with origin at the top left of the printable page area.

Returns

0 if OK, non-zero if any error

Reimplemented from Fl_Paged_Device.

void Fl_Printer::translate (int *x*, int *y*) `[virtual]`

Translates the current graphics origin accounting for the current rotation.

This function is only useful after a rotate() call. Each translate() call must be matched by an untranslate() call. Successive translate() calls add up their effects.

Reimplemented from Fl_Paged_Device.

The documentation for this class was generated from the following files:

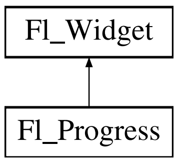
- Fl_Printer.H
- Fl_Printer.cxx

31.105 Fl_Progress Class Reference

Displays a progress bar for the user.

`#include <Fl_Progress.H>`

Inheritance diagram for Fl_Progress:



Public Member Functions

- Fl_Progress (int x, int y, int w, int h, const char ∗l=0)

  The constructor creates the progress bar using the position, size, and label.
- void maximum (float v)

  Sets the maximum value in the progress widget.
- float maximum () const

  Gets the maximum value in the progress widget.
- void minimum (float v)

  Sets the minimum value in the progress widget.
- float minimum () const

  Gets the minimum value in the progress widget.
- void value (float v)

  Sets the current value in the progress widget.
- float value () const

  Gets the current value in the progress widget.

Protected Member Functions

- virtual void draw ()

  Draws the progress bar.

Additional Inherited Members

31.105.1 Detailed Description

Displays a progress bar for the user.

31.105.2 Constructor & Destructor Documentation

Fl_Progress::Fl_Progress (int *X*, int *Y*, int *W*, int *H*, const char ∗ *L* = 0)

The constructor creates the progress bar using the position, size, and label.

You can set the background color with color() and the progress bar color with selection_color(), or you can set both colors together with color(unsigned bg, unsigned sel).

The default colors are FL_BACKGROUND2_COLOR and FL_YELLOW, resp.

31.105.3 Member Function Documentation

void Fl_Progress::draw (void) `[protected]`, `[virtual]`

Draws the progress bar.

Implements Fl_Widget.

void Fl_Progress::maximum (float *v*) `[inline]`

Sets the maximum value in the progress widget.

float Fl_Progress::maximum ( ) const `[inline]`

Gets the maximum value in the progress widget.

void Fl_Progress::minimum (float *v*) `[inline]`

Sets the minimum value in the progress widget.

float Fl_Progress::minimum ( ) const `[inline]`

Gets the minimum value in the progress widget.

void Fl_Progress::value (float *v*) `[inline]`

Sets the current value in the progress widget.

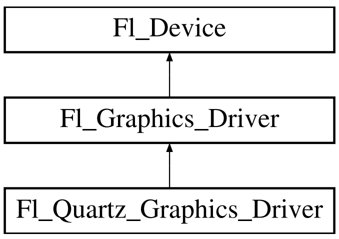
float Fl_Progress::value ( ) const `[inline]`

Gets the current value in the progress widget.
The documentation for this class was generated from the following files:

- Fl_Progress.H
- Fl_Progress.cxx

31.106 Fl_Quartz_Graphics_Driver Class Reference

The Mac OS X-specific graphics class.
`#include <Fl_Device.H>`
Inheritance diagram for Fl_Quartz_Graphics_Driver:



Public Member Functions

- const char ∗ class_name ()

  Returns the name of the class of this object.
- void color (Fl_Color c)

  see fl_color(Fl_Color c).
- void color (uchar r, uchar g, uchar b)

  see fl_color(uchar r, uchar g, uchar b).
- void copy_offscreen (int x, int y, int w, int h, Fl_Offscreen pixmap, int srcx, int srcy)

  see fl_copy_offscreen()
- int descent ()

  see fl_descent().
- void draw (const char ∗str, int n, int x, int y)

  see fl_draw(const char ∗str, int n, int x, int y).
- void draw (int angle, const char ∗str, int n, int x, int y)

  see fl_draw(int angle, const char ∗str, int n, int x, int y).
- void draw (Fl_Pixmap ∗pxm, int XP, int YP, int WP, int HP, int cx, int cy)

Draws an Fl_Pixmap object to the device.

- void draw (Fl_Bitmap ∗pxm, int XP, int YP, int WP, int HP, int cx, int cy)

  Draws an Fl_Bitmap object to the device.
- void draw (Fl_RGB_Image ∗img, int XP, int YP, int WP, int HP, int cx, int cy)

  Draws an Fl_RGB_Image object to the device.
- void draw_image (const uchar ∗buf, int X, int Y, int W, int H, int D=3, int L=0)

  see fl_draw_image(const uchar∗ buf, int X,int Y,int W,int H, int D, int L).
- void draw_image (Fl_Draw_Image_Cb cb, void ∗data, int X, int Y, int W, int H, int D=3)

  see fl_draw_image(Fl_Draw_Image_Cb cb, void∗ data, int X,int Y,int W,int H, int D).
- void draw_image_mono (const uchar ∗buf, int X, int Y, int W, int H, int D=1, int L=0)

  see fl_draw_image_mono(const uchar∗ buf, int X,int Y,int W,int H, int D, int L).
- void draw_image_mono (Fl_Draw_Image_Cb cb, void ∗data, int X, int Y, int W, int H, int D=1)

  see fl_draw_image_mono(Fl_Draw_Image_Cb cb, void∗ data, int X,int Y,int W,int H, int D).
- int draw_scaled (Fl_Image ∗img, int XP, int YP, int WP, int HP)

  Draws an Fl_Image scaled to width `W` & height `H` with top-left corner at X,Y.
- void font (Fl_Font face, Fl_Fontsize size)

  see fl_font(Fl_Font face, Fl_Fontsize size).
- int height ()

  see fl_height().
- void rtl_draw (const char ∗str, int n, int x, int y)

  see fl_rtl_draw(const char ∗str, int n, int x, int y).
- void text_extents (const char ∗, int n, int &dx, int &dy, int &w, int &h)

  see fl_text_extents(const char∗, int n, int& dx, int& dy, int& w, int& h).
- double width (const char ∗str, int n)

  see fl_width(const char ∗str, int n).
- double width (unsigned int c)

  see fl_width(unsigned int n).

Static Public Attributes

- static const char ∗ **class_id** = "Fl_Quartz_Graphics_Driver"

Additional Inherited Members

31.106.1 Detailed Description

The Mac OS X-specific graphics class.

This class is implemented only on the Mac OS X platform.

31.106.2 Member Function Documentation

const char∗ Fl_Quartz_Graphics_Driver::class_name ( ) `[inline]`, `[virtual]`

Returns the name of the class of this object.

Use of the class_name() function is discouraged because it will be removed from future FLTK versions.

The class of an instance of an Fl_Device subclass can be checked with code such as:

```
if ( instance->class_name() == Fl_Printer::class_id ) { ... }
```

Reimplemented from Fl_Graphics_Driver.

void Fl_Quartz_Graphics_Driver::color (Fl_Color *c*) `[virtual]`

see fl_color(Fl_Color c).

Reimplemented from Fl_Graphics_Driver.

void Fl_Quartz_Graphics_Driver::color (uchar *r*, uchar *g*, uchar *b*) `[virtual]`

see fl_color(uchar r, uchar g, uchar b).

Reimplemented from Fl_Graphics_Driver.

int Fl_Quartz_Graphics_Driver::descent ( ) `[virtual]`

see fl_descent().

Reimplemented from Fl_Graphics_Driver.

void Fl_Quartz_Graphics_Driver::draw (const char ∗ *str*, int *n*, int *x*, int *y*) `[virtual]`

see fl_draw(const char ∗str, int n, int x, int y).

Reimplemented from Fl_Graphics_Driver.

void Fl_Quartz_Graphics_Driver::draw (int *angle*, const char ∗ *str*, int *n*, int *x*, int *y*) `[virtual]`

see fl_draw(int angle, const char ∗str, int n, int x, int y).

Reimplemented from Fl_Graphics_Driver.

void Fl_Quartz_Graphics_Driver::draw (Fl_Pixmap ∗ *pxm*, int *XP*, int *YP*, int *WP*, int *HP*, int *cx*, int *cy*) `[virtual]`

Draws an Fl_Pixmap object to the device.

Specifies a bounding box for the image, with the origin (upper left-hand corner) of the image offset by the cx and cy arguments.

Reimplemented from Fl_Graphics_Driver.

void Fl_Quartz_Graphics_Driver::draw (Fl_Bitmap ∗ *bm*, int *XP*, int *YP*, int *WP*, int *HP*, int *cx*, int *cy*) `[virtual]`

Draws an Fl_Bitmap object to the device.

Specifies a bounding box for the image, with the origin (upper left-hand corner) of the image offset by the cx and cy arguments.

Reimplemented from Fl_Graphics_Driver.

void Fl_Quartz_Graphics_Driver::draw (Fl_RGB_Image ∗ *rgb*, int *XP*, int *YP*, int *WP*, int *HP*, int *cx*, int *cy*) `[virtual]`

Draws an Fl_RGB_Image object to the device.

Specifies a bounding box for the image, with the origin (upper left-hand corner) of the image offset by the cx and cy arguments.

Reimplemented from Fl_Graphics_Driver.

void Fl_Quartz_Graphics_Driver::draw_image (const uchar ∗ *buf*, int *X*, int *Y*, int *W*, int *H*, int *D* = *3*, int *L* = *0*) `[virtual]`

see fl_draw_image(const uchar∗ buf, int X,int Y,int W,int H, int D, int L).

Reimplemented from Fl_Graphics_Driver.

void Fl_Quartz_Graphics_Driver::draw_image (Fl_Draw_Image_Cb *cb,* void ∗ *data,* int *X,* int *Y,* int *W,* int *H,* int *D* = *3*) `[virtual]`

see fl_draw_image(Fl_Draw_Image_Cb cb, void∗ data, int X,int Y,int W,int H, int D).
Reimplemented from Fl_Graphics_Driver.

void Fl_Quartz_Graphics_Driver::draw_image_mono (const uchar ∗ *buf,* int *X,* int *Y,* int *W,* int *H,* int *D* = *1,* int *L* = *0*) `[virtual]`

see fl_draw_image_mono(const uchar∗ buf, int X,int Y,int W,int H, int D, int L).
Reimplemented from Fl_Graphics_Driver.

void Fl_Quartz_Graphics_Driver::draw_image_mono (Fl_Draw_Image_Cb *cb,* void ∗ *data,* int *X,* int *Y,* int *W,* int *H,* int *D* = *1*) `[virtual]`

see fl_draw_image_mono(Fl_Draw_Image_Cb cb, void∗ data, int X,int Y,int W,int H, int D).
Reimplemented from Fl_Graphics_Driver.

int Fl_Quartz_Graphics_Driver::draw_scaled (Fl_Image ∗ *img,* int *X,* int *Y,* int *W,* int *H*) `[virtual]`

Draws an Fl_Image scaled to width `W` & height `H` with top-left corner at *X*,Y.

Returns

zero when the graphics driver doesn't implement scaled drawing, non-zero if it does implement it.

Reimplemented from Fl_Graphics_Driver.

void Fl_Quartz_Graphics_Driver::font (Fl_Font *face,* Fl_Fontsize *fsize*) `[virtual]`

see fl_font(Fl_Font face, Fl_Fontsize size).
Reimplemented from Fl_Graphics_Driver.

int Fl_Quartz_Graphics_Driver::height ( ) `[virtual]`

see fl_height().
Reimplemented from Fl_Graphics_Driver.

void Fl_Quartz_Graphics_Driver::rtl_draw (const char ∗ *str,* int *n,* int *x,* int *y*) `[virtual]`

see fl_rtl_draw(const char ∗str, int n, int x, int y).
Reimplemented from Fl_Graphics_Driver.

void Fl_Quartz_Graphics_Driver::text_extents (const char ∗ *t,* int *n,* int & *dx,* int & *dy,* int & *w,* int & *h*) `[virtual]`

see fl_text_extents(const char∗, int n, int& dx, int& dy, int& w, int& h).
Reimplemented from Fl_Graphics_Driver.

double Fl_Quartz_Graphics_Driver::width (const char ∗ *str,* int *n*) `[virtual]`

see fl_width(const char ∗str, int n).
Reimplemented from Fl_Graphics_Driver.

double Fl_Quartz_Graphics_Driver::width (unsigned int *c*) `[virtual]`

see fl_width(unsigned int n).

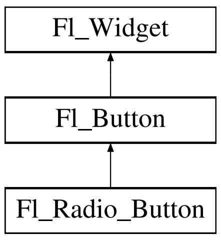
Reimplemented from Fl_Graphics_Driver.

The documentation for this class was generated from the following files:

- Fl_Device.H
- fl_color_mac.cxx
- Fl_Device.cxx
- Fl_Double_Window.cxx
- fl_draw_image_mac.cxx

31.107 Fl_Radio_Button Class Reference

Inheritance diagram for Fl_Radio_Button:



Public Member Functions

- Fl_Radio_Button (int X, int Y, int W, int H, const char ∗L=0)

  The constructor creates the button using the given position, size, and label.

Additional Inherited Members

31.107.1 Constructor & Destructor Documentation

Fl_Radio_Button::Fl_Radio_Button (int *X*, int *Y*, int *W*, int *H*, const char ∗ *L* = `0`)

The constructor creates the button using the given position, size, and label.

The Button type() is set to FL_RADIO_BUTTON.

Parameters

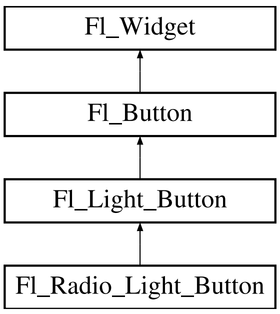
`in`	*X,Y,W,H*	position and size of the widget
`in`	*L*	widget label, default is no label

The documentation for this class was generated from the following files:

- Fl_Radio_Button.H
- Fl_Button.cxx

31.108 Fl_Radio_Light_Button Class Reference

Inheritance diagram for Fl_Radio_Light_Button:



Public Member Functions

- **Fl_Radio_Light_Button** (int X, int Y, int W, int H, const char ∗l=0)

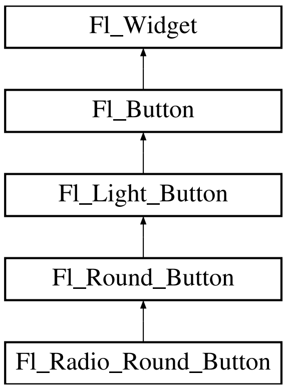
Additional Inherited Members

The documentation for this class was generated from the following files:

- Fl_Radio_Light_Button.H
- Fl_Light_Button.cxx

31.109 Fl_Radio_Round_Button Class Reference

Inheritance diagram for Fl_Radio_Round_Button:



Public Member Functions

- Fl_Radio_Round_Button (int X, int Y, int W, int H, const char ∗L=0)

    Creates a new Fl_Radio_Button widget using the given position, size, and label string.

Additional Inherited Members

31.109.1 Constructor & Destructor Documentation

Fl_Radio_Round_Button::Fl_Radio_Round_Button (int *X*, int *Y*, int *W*, int *H*, const char ∗ *L* = 0)

Creates a new Fl_Radio_Button widget using the given position, size, and label string.

The button type() is set to FL_RADIO_BUTTON.

Parameters

in	*X,Y,W,H*	position and size of the widget
in	*L*	widget label, default is no label

The documentation for this class was generated from the following files:

- Fl_Radio_Round_Button.H
- Fl_Round_Button.cxx

31.110 Fl_Scroll::ScrollInfo::Fl_Region_LRTB Struct Reference

A local struct to manage a region defined by left/right/top/bottom.

`#include <Fl_Scroll.H>`

Public Attributes

- int b

  (b)ottom "y" position, aka y2
- int l

  (l)eft "x" position, aka x1
- int r

  (r)ight "x" position, aka x2
- int t

  (t)op "y" position, aka y1

31.110.1 Detailed Description

A local struct to manage a region defined by left/right/top/bottom.

The documentation for this struct was generated from the following file:

- Fl_Scroll.H

31.111 Fl_Scroll::ScrollInfo::Fl_Region_XYWH Struct Reference

A local struct to manage a region defined by xywh.

`#include <Fl_Scroll.H>`

Public Attributes

- int **h**
- int **w**
- int **x**
- int **y**

31.111.1 Detailed Description

A local struct to manage a region defined by xywh.

The documentation for this struct was generated from the following file:

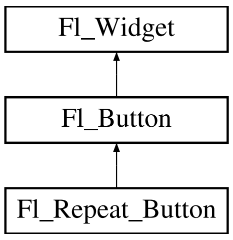
- Fl_Scroll.H

31.112 Fl_Repeat_Button Class Reference

The Fl_Repeat_Button is a subclass of Fl_Button that generates a callback when it is pressed and then repeatedly generates callbacks as long as it is held down.

`#include <Fl_Repeat_Button.H>`

Inheritance diagram for Fl_Repeat_Button:



Public Member Functions

- void **deactivate** ()
- Fl_Repeat_Button (int X, int Y, int W, int H, const char ∗l=0)

  Creates a new Fl_Repeat_Button widget using the given position, size, and label string.
- int handle (int)

  Handles the specified event.

Additional Inherited Members

31.112.1 Detailed Description

The Fl_Repeat_Button is a subclass of Fl_Button that generates a callback when it is pressed and then repeatedly generates callbacks as long as it is held down.

The speed of the repeat is fixed and depends on the implementation.

31.112.2 Constructor & Destructor Documentation

Fl_Repeat_Button::Fl_Repeat_Button (int *X*, int *Y*, int *W*, int *H*, const char ∗ *l* = 0)

Creates a new Fl_Repeat_Button widget using the given position, size, and label string.

The default boxtype is FL_UP_BOX. Deletes the button.

31.112.3 Member Function Documentation

int Fl_Repeat_Button::handle (int *event*) `[virtual]`

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited handle() method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters

in	*event*	the kind of event received

Return values

0	if the event was not used or understood
1	if the event was used and can be deleted

See Also

Fl_Event

Reimplemented from Fl_Button.
The documentation for this class was generated from the following files:

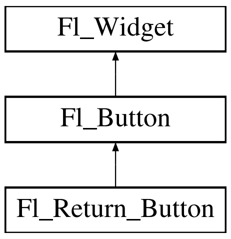
- Fl_Repeat_Button.H
- Fl_Repeat_Button.cxx

31.113 Fl_Return_Button Class Reference

The Fl_Return_Button is a subclass of Fl_Button that generates a callback when it is pressed or when the user presses the Enter key.

`#include <Fl_Return_Button.H>`

Inheritance diagram for Fl_Return_Button:



Public Member Functions

- Fl_Return_Button (int X, int Y, int W, int H, const char ∗l=0)

  Creates a new Fl_Return_Button widget using the given position, size, and label string.
- int handle (int)

  Handles the specified event.

Protected Member Functions

- void draw ()

  Draws the widget.

Additional Inherited Members

31.113.1 Detailed Description

The Fl_Return_Button is a subclass of Fl_Button that generates a callback when it is pressed or when the user presses the Enter key.

A carriage-return symbol is drawn next to the button label.

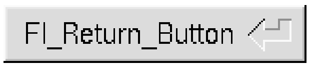


Figure 31.25: Fl_Return_Button

31.113.2 Constructor & Destructor Documentation

Fl_Return_Button::Fl_Return_Button (int *X*, int *Y*, int *W*, int *H*, const char ∗ *l* = `0`)

Creates a new Fl_Return_Button widget using the given position, size, and label string.
The default boxtype is FL_UP_BOX.
The inherited destructor deletes the button.

31.113.3 Member Function Documentation

void Fl_Return_Button::draw ( ) `[protected]`, `[virtual]`

Draws the widget.
Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call redraw() instead.
Override this function to draw your own widgets.
If you ever need to call another widget's draw method *from within your own draw() method*, e.g. for an embedded scrollbar, you can do it (because draw() is virtual) like this:

```
Fl_Widget *s = &scroll;                    // scroll is an embedded Fl_Scrollbar
s->draw();                       // calls Fl_Scrollbar::draw()
```

Reimplemented from Fl_Button.

int Fl_Return_Button::handle (int *event*) `[virtual]`

Handles the specified event.
You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.
When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.
Most of the time, you want to call the inherited handle() method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.
Parameters

`in`	*event*	the kind of event received

Return values

0	if the event was not used or understood
1	if the event was used and can be deleted

See Also

Fl_Event

Reimplemented from Fl_Button.
The documentation for this class was generated from the following files:

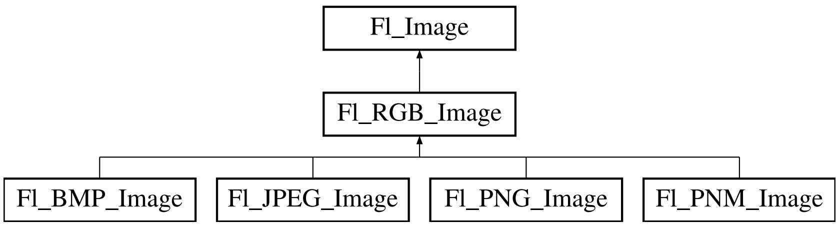
- Fl_Return_Button.H
- Fl_Return_Button.cxx

31.114 Fl_RGB_Image Class Reference

The Fl_RGB_Image class supports caching and drawing of full-color images with 1 to 4 channels of color information.

`#include <Fl_Image.H>`

Inheritance diagram for Fl_RGB_Image:



Public Member Functions

- virtual void color_average (Fl_Color c, float i)

  The color_average() method averages the colors in the image with the FLTK color value c.
- virtual Fl_Image ∗ copy (int W, int H)

  The copy() method creates a copy of the specified image.
- Fl_Image ∗ **copy** ()
- virtual void desaturate ()

  The desaturate() method converts an image to grayscale.
- virtual void draw (int X, int Y, int W, int H, int cx=0, int cy=0)

  Draws the image with a bounding box.
- void **draw** (int X, int Y)
- Fl_RGB_Image (const uchar ∗bits, int W, int H, int D=3, int LD=0)

  The constructor creates a new image from the specified data.
- Fl_RGB_Image (const Fl_Pixmap ∗pxm, Fl_Color bg=FL_GRAY)

  The constructor creates a new RGBA image from the specified Fl_Pixmap.
- virtual void label (Fl_Widget ∗w)

  The label() methods are an obsolete way to set the image attribute of a widget or menu item.
- virtual void label (Fl_Menu_Item ∗m)

  The label() methods are an obsolete way to set the image attribute of a widget or menu item.
- virtual void uncache ()

  If the image has been cached for display, delete the cache data.
- virtual ∼Fl_RGB_Image ()

  The destructor frees all memory and server resources that are used by the image.

Static Public Member Functions

- static void max_size (size_t size)

  Sets the maximum allowed image size in bytes when creating an Fl_RGB_Image object.
- static size_t max_size ()

  Returns the maximum allowed image size in bytes when creating an Fl_RGB_Image object.

Public Attributes

- int alloc_array

  If non-zero, the object's data array is delete[]'d when deleting the object.
- const uchar ∗ array

  Points to the start of the object's data array.

Friends

- class **Fl_GDI_Graphics_Driver**
- class **Fl_GDI_Printer_Graphics_Driver**
- class **Fl_Quartz_Graphics_Driver**
- class **Fl_Xlib_Graphics_Driver**

Additional Inherited Members

31.114.1 Detailed Description

The Fl_RGB_Image class supports caching and drawing of full-color images with 1 to 4 channels of color information.

Images with an even number of channels are assumed to contain alpha information, which is used to blend the image with the contents of the screen.

Fl_RGB_Image is defined in <FL/Fl_Image.H>, however for compatibility reasons <FL/Fl_RGB_-Image.H> should be included.

31.114.2 Constructor & Destructor Documentation

Fl_RGB_Image::Fl_RGB_Image (const uchar ∗ *bits*, int *W*, int *H*, int *D* = *3*, int *LD* = *0*)

The constructor creates a new image from the specified data.

The data array `bits` must contain sufficient data to provide `W * H * D` image bytes and optional line padding, see `LD`.

`W` and `H` are the width and height of the image in pixels, resp.

`D` is the image depth and can be:

- D=1: each uchar in `bits`[] is a grayscale pixel value
- D=2: each uchar pair in `bits`[] is a grayscale + alpha pixel value
- D=3: each uchar triplet in `bits`[] is an R/G/B pixel value
- D=4: each uchar quad in `bits`[] is an R/G/B/A pixel value

`LD` specifies the line data size of the array, see Fl_Image::ld(int). If `LD` is zero, then `W * D` is assumed, otherwise `LD` must be greater than or equal to `W * D` to account for (unused) extra data per line (padding).

The caller is responsible that the image data array `bits` persists as long as the image is used.

This constructor sets Fl_RGB_Image::alloc_array to 0. To have the image object control the deallocation of the data array `bits`, set alloc_array to non-zero after construction.

Parameters

`in`	*bits*	The image data array.

in	*W*	The width of the image in pixels.
in	*H*	The height of the image in pixels.
in	*D*	The image depth, or 'number of channels' (default=3).
in	*LD*	Line data size (default=0).

See Also

Fl_Image::data(), Fl_Image::w(), Fl_Image::h(), Fl_Image::d(), Fl_Image::ld(int)

Fl_RGB_Image::Fl_RGB_Image (const Fl_Pixmap ∗ *pxm*, Fl_Color *bg* = *FL_GRAY*)

The constructor creates a new RGBA image from the specified Fl_Pixmap.

The RGBA image is built fully opaque except for the transparent area of the pixmap that is assigned the `bg` color with full transparency.

This constructor creates a new internal data array and sets Fl_RGB_Image::alloc_array to 1 so the data array is deleted when the image is destroyed.

31.114.3 Member Function Documentation

void Fl_RGB_Image::color_average (Fl_Color *c*, float *i*) `[virtual]`

The color_average() method averages the colors in the image with the FLTK color value c.

The i argument specifies the amount of the original image to combine with the color, so a value of 1.0 results in no color blend, and a value of 0.0 results in a constant image of the specified color.

An internal copy is made of the original image before changes are applied, to avoid modifying the original image.

Reimplemented from Fl_Image.

Fl_Image ∗ Fl_RGB_Image::copy (int *W*, int *H*) `[virtual]`

The copy() method creates a copy of the specified image.

If the width and height are provided, the image is resized to the specified size. The image should be deleted (or in the case of Fl_Shared_Image, released) when you are done with it.

Reimplemented from Fl_Image.

void Fl_RGB_Image::desaturate ( ) `[virtual]`

The desaturate() method converts an image to grayscale.

If the image contains an alpha channel (depth = 4), the alpha channel is preserved.

An internal copy is made of the original image before changes are applied, to avoid modifying the original image.

Reimplemented from Fl_Image.

void Fl_RGB_Image::draw (int *X*, int *Y*, int *W*, int *H*, int *cx* = *0*, int *cy* = *0*) `[virtual]`

Draws the image with a bounding box.

Arguments `X,Y,W,H` specify a bounding box for the image, with the origin (upper-left corner) of the image offset by the `cx` and `cy` arguments.

In other words: `fl_push_clip(X,Y,W,H)` is applied, the image is drawn with its upper-left corner at `X-cx,Y-cy` and its own width and height, `fl_pop_clip()` is applied.

Reimplemented from Fl_Image.

void Fl_RGB_Image::label (Fl_Widget * *widget*) `[virtual]`

The label() methods are an obsolete way to set the image attribute of a widget or menu item.

Use the image() or deimage() methods of the Fl_Widget and Fl_Menu_Item classes instead.

Reimplemented from Fl_Image.

void Fl_RGB_Image::label (Fl_Menu_Item * *m*) `[virtual]`

The label() methods are an obsolete way to set the image attribute of a widget or menu item.

Use the image() or deimage() methods of the Fl_Widget and Fl_Menu_Item classes instead.

Reimplemented from Fl_Image.

static void Fl_RGB_Image::max_size (size_t *size*) `[inline], [static]`

Sets the maximum allowed image size in bytes when creating an Fl_RGB_Image object.

The image size in bytes of an Fl_RGB_Image object is the value of the product w() * h() * d(). If this product exceeds size, the created object of a derived class of Fl_RGB_Image won't be loaded with the image data. This does not apply to direct RGB image creation with Fl_RGB_Image::Fl_RGB_Image(const uchar *bits, int W, int H, int D, int LD). The default max_size() value is essentially infinite.

static size_t Fl_RGB_Image::max_size ( ) `[inline], [static]`

Returns the maximum allowed image size in bytes when creating an Fl_RGB_Image object.

See Also

void Fl_RGB_Image::max_size(size_t)

void Fl_RGB_Image::uncache ( ) `[virtual]`

If the image has been cached for display, delete the cache data.

This allows you to change the data used for the image and then redraw it without recreating an image object.

Reimplemented from Fl_Image.

The documentation for this class was generated from the following files:

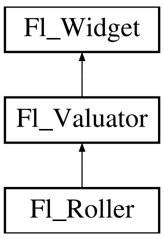
- Fl_Image.H
- Fl_Image.cxx

31.115 Fl_Roller Class Reference

The Fl_Roller widget is a "dolly" control commonly used to move 3D objects.

`#include <Fl_Roller.H>`

Inheritance diagram for Fl_Roller:



Public Member Functions

- Fl_Roller (int X, int Y, int W, int H, const char ∗L=0)

  Creates a new Fl_Roller widget using the given position, size, and label string.
- int handle (int)

  Handles the specified event.

Protected Member Functions

- void draw ()

  Draws the widget.

Additional Inherited Members

31.115.1 Detailed Description

The Fl_Roller widget is a ”dolly” control commonly used to move 3D objects.

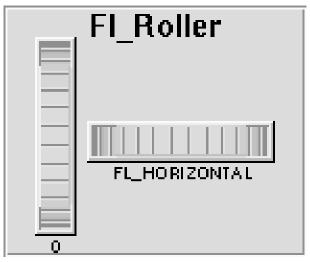


Figure 31.26: Fl_Roller

31.115.2 Constructor & Destructor Documentation

Fl_Roller::Fl_Roller (int *X*, int *Y*, int *W*, int *H*, const char ∗ *L* = `0`)

Creates a new Fl_Roller widget using the given position, size, and label string.
The default boxtype is FL_NO_BOX.
Inherited destructor destroys the valuator.

31.115.3 Member Function Documentation

void Fl_Roller::draw ( ) `[protected]`, `[virtual]`

Draws the widget.
Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call redraw() instead.
Override this function to draw your own widgets.
If you ever need to call another widget's draw method *from within your own draw() method*, e.g. for an embedded scrollbar, you can do it (because draw() is virtual) like this:

```
Fl_Widget *s = &scroll;                // scroll is an embedded Fl_Scrollbar
s->draw();                      // calls Fl_Scrollbar::draw()
```

Implements Fl_Widget.

int Fl_Roller::handle (int *event*) `[virtual]`

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited handle() method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters

`in`	*event*	the kind of event received

Return values

0	if the event was not used or understood
1	if the event was used and can be deleted

See Also

Fl_Event

Reimplemented from Fl_Widget.

The documentation for this class was generated from the following files:

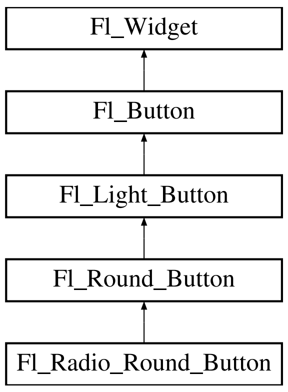
- Fl_Roller.H
- Fl_Roller.cxx

31.116 Fl_Round_Button Class Reference

Buttons generate callbacks when they are clicked by the user.

`#include <Fl_Round_Button.H>`

Inheritance diagram for Fl_Round_Button:



Public Member Functions

- Fl_Round_Button (int x, int y, int w, int h, const char ∗l=0)

  Creates a new Fl_Round_Button widget using the given position, size, and label string.

Additional Inherited Members

31.116.1 Detailed Description

Buttons generate callbacks when they are clicked by the user.

You control exactly when and how by changing the values for type() and when().

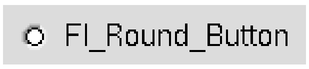


Figure 31.27: Fl_Round_Button

The Fl_Round_Button subclass display the "on" state by turning on a light, rather than drawing pushed in. The shape of the "light" is initially set to FL_ROUND_DOWN_BOX. The color of the light when on is controlled with selection_color(), which defaults to FL_FOREGROUND_COLOR.

31.116.2 Constructor & Destructor Documentation

Fl_Round_Button::Fl_Round_Button (int *X*, int *Y*, int *W*, int *H*, const char ∗ *L* = `0`)

Creates a new Fl_Round_Button widget using the given position, size, and label string.

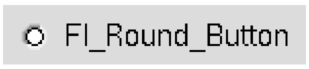


Figure 31.28: Fl_Round_Button

The Fl_Round_Button subclass displays the "ON" state by turning on a light, rather than drawing pushed in.

The default box type is FL_NO_BOX, which draws the label w/o a box right of the checkmark.

The shape of the "light" is set with down_box() and its default value is FL_ROUND_DOWN_BOX.

The color of the light when on is controlled with selection_color(), which defaults to FL_FOREGROUND_COLOR (usually black).

Parameters

`in`	*X,Y,W,H*	position and size of the widget
`in`	*L*	widget label, default is no label

The documentation for this class was generated from the following files:

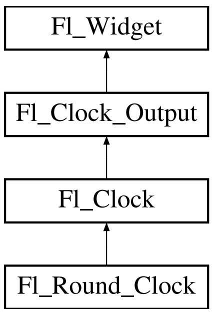
- Fl_Round_Button.H
- Fl_Round_Button.cxx

31.117 Fl_Round_Clock Class Reference

A clock widget of type FL_ROUND_CLOCK.

`#include <Fl_Round_Clock.H>`

Inheritance diagram for Fl_Round_Clock:



Public Member Functions

- Fl_Round_Clock (int X, int Y, int W, int H, const char *L=0)

  Creates the clock widget, setting his type and box.

Additional Inherited Members

31.117.1 Detailed Description

A clock widget of type FL_ROUND_CLOCK.

Has no box.

31.117.2 Constructor & Destructor Documentation

Fl_Round_Clock::Fl_Round_Clock (int *X*, int *Y*, int *W*, int *H*, const char * *L* = 0)

Creates the clock widget, setting his type and box.

The documentation for this class was generated from the following files:

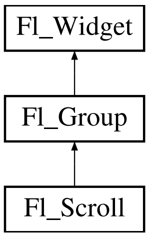
- Fl_Round_Clock.H
- Fl_Clock.cxx

31.118 Fl_Scroll Class Reference

This container widget lets you maneuver around a set of widgets much larger than your window.

`#include <Fl_Scroll.H>`

Inheritance diagram for Fl_Scroll:



Public Types

- enum {
  HORIZONTAL = 1, **VERTICAL** = 2, **BOTH** = 3, **ALWAYS_ON** = 4,
  HORIZONTAL_ALWAYS = 5, **VERTICAL_ALWAYS** = 6, **BOTH_ALWAYS** = 7 }

Public Member Functions

- void clear ()

  Clear all but the scrollbars...
- Fl_Scroll (int X, int Y, int W, int H, const char *l=0)

  Creates a new Fl_Scroll widget using the given position, size, and label string.
- int handle (int)

  Handles the specified event.
- void resize (int X, int Y, int W, int H)

  Resizes the Fl_Scroll widget and moves its children if necessary.
- void scroll_to (int, int)

  Moves the contents of the scroll group to a new position.
- int scrollbar_size () const

  Gets the current size of the scrollbars' troughs, in pixels.
- void scrollbar_size (int newSize)

  Sets the pixel size of the scrollbars' troughs to `newSize`, in pixels.
- int xposition () const

  Gets the current horizontal scrolling position.
- int yposition () const

  Gets the current vertical scrolling position.

Public Attributes

- Fl_Scrollbar **hscrollbar**
- Fl_Scrollbar **scrollbar**

Protected Member Functions

- void bbox (int &, int &, int &, int &)

  Returns the bounding box for the interior of the scrolling area, inside the scrollbars.
- void draw ()

  Draws the widget.

Additional Inherited Members

31.118.1 Detailed Description

This container widget lets you maneuver around a set of widgets much larger than your window.

If the child widgets are larger than the size of this object then scrollbars will appear so that you can scroll over to them:

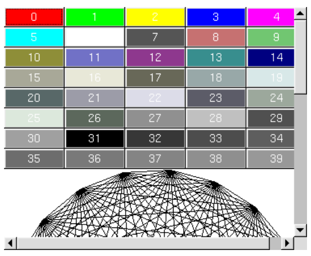


Figure 31.29: Fl_Scroll

If all of the child widgets are packed together into a solid rectangle then you want to set box() to FL_NO_BOX or one of the _FRAME types. This will result in the best output. However, if the child widgets are a sparse arrangement you must set box() to a real _BOX type. This can result in some blinking during redrawing, but that can be solved by using a Fl_Double_Window.

By default you can scroll in both directions, and the scrollbars disappear if the data will fit in the area of the scroll.

Use Fl_Scroll::type() to change this as follows :

- 0 - No scrollbars
- Fl_Scroll::HORIZONTAL - Only a horizontal scrollbar.
- Fl_Scroll::VERTICAL - Only a vertical scrollbar.
- Fl_Scroll::BOTH - The default is both scrollbars.
- Fl_Scroll::HORIZONTAL_ALWAYS - Horizontal scrollbar always on, vertical always off.
- Fl_Scroll::VERTICAL_ALWAYS - Vertical scrollbar always on, horizontal always off.
- Fl_Scroll::BOTH_ALWAYS - Both always on.

Use **scrollbar.align(int) (see void Fl_Widget::align(Fl_Align)) :** to change what side the scrollbars are drawn on.

If the FL_ALIGN_LEFT bit is on, the vertical scrollbar is on the left. If the FL_ALIGN_TOP bit is on, the horizontal scrollbar is on the top. Note that only the alignment flags in scrollbar are considered. The flags in hscrollbar however are ignored.

This widget can also be used to pan around a single child widget "canvas". This child widget should be of your own class, with a draw() method that draws the contents. The scrolling is done by changing the x() and y() of the widget, so this child must use the x() and y() to position its drawing. To speed up drawing it should test fl_not_clipped(int x,int y,int w,int h) to find out if a particular area of the widget must be drawn.

Another very useful child is a single Fl_Pack, which is itself a group that packs its children together and changes size to surround them. Filling the Fl_Pack with Fl_Tabs groups (and then putting normal widgets inside those) gives you a very powerful scrolling list of individually-openable panels.

Fluid lets you create these, but you can only lay out objects that fit inside the Fl_Scroll without scrolling. Be sure to leave space for the scrollbars, as Fluid won't show these either.

You cannot use Fl_Window as a child of this since the clipping is not conveyed to it when drawn, and it will draw over the scrollbars and neighboring objects.

31.118.2 Constructor & Destructor Documentation

Fl_Scroll::Fl_Scroll (int *X*, int *Y*, int *W*, int *H*, const char * *L* = `0`)

Creates a new Fl_Scroll widget using the given position, size, and label string.

The default boxtype is FL_NO_BOX.

The destructor *also deletes all the children*. This allows a whole tree to be deleted at once, without having to keep a pointer to all the children in the user code. A kludge has been done so the Fl_Scroll and all of its children can be automatic (local) variables, but you must declare the Fl_Scroll *first*, so that it is destroyed last.

31.118.3 Member Function Documentation

void Fl_Scroll::bbox (int & *X*, int & *Y*, int & *W*, int & *H*) `[protected]`

Returns the bounding box for the interior of the scrolling area, inside the scrollbars.

Currently this is only reliable after draw(), and before any resizing of the Fl_Scroll or any child widgets occur.

Todo The visibility of the scrollbars ought to be checked/calculated outside of the draw() method (STR #1895).

void Fl_Scroll::clear ( )

Clear all but the scrollbars...

void Fl_Scroll::draw ( ) [protected], [virtual]

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call redraw() instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw() method*, e.g. for an embedded scrollbar, you can do it (because draw() is virtual) like this:

```
Fl_Widget *s = &scroll;              // scroll is an embedded Fl_Scrollbar
s->draw();                     // calls Fl_Scrollbar::draw()
```

Reimplemented from Fl_Group.

int Fl_Scroll::handle (int *event*) [virtual]

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited handle() method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters

in	*event*	the kind of event received

Return values

0	if the event was not used or understood
1	if the event was used and can be deleted

See Also

Fl_Event

Reimplemented from Fl_Group.

void Fl_Scroll::resize (int *X*, int *Y*, int *W*, int *H*) [virtual]

Resizes the Fl_Scroll widget and moves its children if necessary.

The Fl_Scroll widget first resizes itself, and then it moves all its children if (and only if) the Fl_Scroll widget has been moved. The children are moved by the same amount as the Fl_Scroll widget has been moved, hence all children keep their relative positions.

Note

Fl_Scroll::resize() does **not** call Fl_Group::resize(), and child widgets are **not** resized.

Since children of an Fl_Scroll are not resized, the resizable() widget is ignored (if it is set).

The scrollbars are moved to their proper positions, as given by Fl_Scroll::scrollbar.align(), and switched on or off as necessary.

Note

Due to current (FLTK 1.3.x) implementation constraints some of this may effectively be postponed until the Fl_Scroll is drawn the next time. This may change in a future release.

See Also

Fl_Group::resizable()
Fl_Widget::resize(int,int,int,int)

Reimplemented from Fl_Group.

void Fl_Scroll::scroll_to (int *X*, int *Y*)

Moves the contents of the scroll group to a new position.

This is like moving the scrollbars of the Fl_Scroll around. For instance:

```
Fl_Scroll scroll (10,10,200,200);
Fl_Box b1 ( 10, 10,50,50,"b1"); // relative (x,y) = (0,0)
Fl_Box b2 ( 60, 60,50,50,"b2"); // relative (x,y) = (50,50)
Fl_Box b3 ( 60,110,50,50,"b3"); // relative (x,y) = (50,100)
// populate scroll with more children ...
scroll.end();
scroll.scroll_to(50,100);
```

will move the logical origin of the internal scroll area to (-50,-100) relative to the origin of the Fl_Scroll (10,10), i.e. Fl_Box b3 will be visible in the top left corner of the scroll area.

int Fl_Scroll::scrollbar_size ( ) const `[inline]`

Gets the current size of the scrollbars' troughs, in pixels.

If this value is zero (default), this widget will use the Fl::scrollbar_size() value as the scrollbar's width.

Returns

Scrollbar size in pixels, or 0 if the global Fl::scrollbar_size() is being used.

See Also

Fl::scrollbar_size(int)

void Fl_Scroll::scrollbar_size (int *newSize*) `[inline]`

Sets the pixel size of the scrollbars' troughs to `newSize`, in pixels.

Normally you should not need this method, and should use Fl::scrollbar_size(int) instead to manage the size of ALL your widgets' scrollbars. This ensures your application has a consistent UI, is the default behavior, and is normally what you want.

Only use THIS method if you really need to override the global scrollbar size. The need for this should be rare.

Setting `newSize` to the special value of 0 causes the widget to track the global Fl::scrollbar_size(), which is the default.

Parameters

`in`	*newSize*	Sets the scrollbar size in pixels. If 0 (default), scrollbar size tracks the global Fl::scrollbar_size()

See Also

Fl::scrollbar_size()

int Fl_Scroll::xposition ( ) const `[inline]`

Gets the current horizontal scrolling position.

int Fl_Scroll::yposition ( ) const `[inline]`

Gets the current vertical scrolling position.

The documentation for this class was generated from the following files:

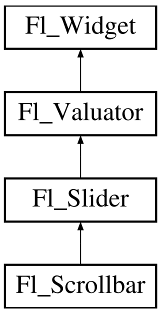
- Fl_Scroll.H
- Fl_Scroll.cxx

31.119 Fl_Scrollbar Class Reference

The Fl_Scrollbar widget displays a slider with arrow buttons at the ends of the scrollbar.

`#include <Fl_Scrollbar.H>`

Inheritance diagram for Fl_Scrollbar:



Public Member Functions

- Fl_Scrollbar (int X, int Y, int W, int H, const char *L=0)

  Creates a new Fl_Scrollbar widget with given position, size, and label.
- int handle (int)

  Handles the specified event.
- int linesize () const

  Get the size of step, in lines, that the arror keys move.
- void linesize (int i)

  This number controls how big the steps are that the arrow keys do.
- int value () const

  Gets the integer value (position) of the slider in the scrollbar.
- int value (int p)

  Sets the value (position) of the slider in the scrollbar.
- int value (int pos, int windowSize, int first, int total)

  Sets the position, size and range of the slider in the scrollbar.
- ∼Fl_Scrollbar ()

  Destroys the Scrollbar.

Protected Member Functions

- void draw ()

  Draws the widget.

Additional Inherited Members

31.119.1 Detailed Description

The Fl_Scrollbar widget displays a slider with arrow buttons at the ends of the scrollbar.

Clicking on the arrows move up/left and down/right by linesize(). Scrollbars also accept FL_SHORTCUT events: the arrows move by linesize(), and vertical scrollbars take Page Up/Down (they move by the page size minus linesize()) and Home/End (they jump to the top or bottom).

Scrollbars have step(1) preset (they always return integers). If desired you can set the step() to non-integer values. You will then have to use casts to get at the floating-point versions of value() from Fl_Slider.

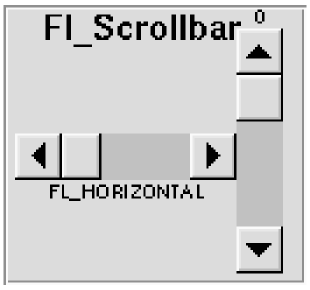


Figure 31.30: Fl_Scrollbar

31.119.2 Constructor & Destructor Documentation

Fl_Scrollbar::Fl_Scrollbar (int *X*, int *Y*, int *W*, int *H*, const char ∗ *L* = 0)

Creates a new Fl_Scrollbar widget with given position, size, and label.

You need to do type(FL_HORIZONTAL) if you want a horizontal scrollbar.

Fl_Scrollbar::∼Fl_Scrollbar ( )

Destroys the Scrollbar.

31.119.3 Member Function Documentation

void Fl_Scrollbar::draw ( ) [protected], [virtual]

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call redraw() instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw() method*, e.g. for an embedded scrollbar, you can do it (because draw() is virtual) like this:

```
Fl_Widget *s = &scroll;                 // scroll is an embedded Fl_Scrollbar
s->draw();                    // calls Fl_Scrollbar::draw()
```

Implements Fl_Widget.

int Fl_Scrollbar::handle (int *event*) [virtual]

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited handle() method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters

in	*event*	the kind of event received

Return values

0	if the event was not used or understood
1	if the event was used and can be deleted

See Also

Fl_Event

Reimplemented from Fl_Widget.

void Fl_Scrollbar::linesize (int *i*) `[inline]`

This number controls how big the steps are that the arrow keys do.

In addition page up/down move by the size last sent to value() minus one linesize(). The default is 16.

int Fl_Scrollbar::value ( ) const `[inline]`

Gets the integer value (position) of the slider in the scrollbar.

You can get the floating point value with Fl_Slider::value().

See Also

Fl_Scrollbar::value(int p)
Fl_Scrollbar::value(int pos, int size, int first, int total)

int Fl_Scrollbar::value (int *p*) `[inline]`

Sets the value (position) of the slider in the scrollbar.

See Also

Fl_Scrollbar::value()
Fl_Scrollbar::value(int pos, int size, int first, int total)

int Fl_Scrollbar::value (int *pos,* int *windowSize,* int *first,* int *total*) `[inline]`

Sets the position, size and range of the slider in the scrollbar.

Parameters

in	*pos*	position, first line displayed
in	*windowSize*	number of lines displayed
in	*first*	number of first line
in	*total*	total number of lines

You should call this every time your window changes size, your data changes size, or your scroll position changes (even if in response to a callback from this scrollbar). All necessary calls to redraw() are done.

Calls Fl_Slider::scrollvalue(int pos, int size, int first, int total).

The documentation for this class was generated from the following files:

- Fl_Scrollbar.H
- Fl_Scrollbar.cxx

31.120 Fl_Scroll::ScrollInfo::Fl_Scrollbar_Data Struct Reference

A local struct to manage a scrollbar's xywh region and tab values.

`#include <Fl_Scroll.H>`

Public Attributes

- int first

  scrollbar tab's "number of first line"
- int **h**
- int pos

  scrollbar tab's "position of first line displayed"
- int size

  scrollbar tab's "size of window in lines"
- int total

  scrollbar tab's "total number of lines"
- int **w**
- int **x**
- int **y**

31.120.1 Detailed Description

A local struct to manage a scrollbar's xywh region and tab values.

The documentation for this struct was generated from the following file:

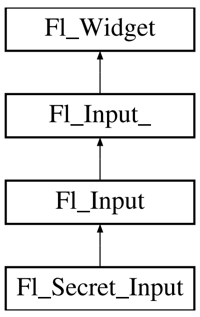
- Fl_Scroll.H

31.121 Fl_Secret_Input Class Reference

The Fl_Secret_Input class is a subclass of Fl_Input that displays its input as a string of placeholders.

`#include <Fl_Secret_Input.H>`

Inheritance diagram for Fl_Secret_Input:



Public Member Functions

- Fl_Secret_Input (int X, int Y, int W, int H, const char ∗l=0)

  Creates a new Fl_Secret_Input widget using the given position, size, and label string.
- int handle (int)

  Handles the specified event.

Additional Inherited Members

31.121.1 Detailed Description

The Fl_Secret_Input class is a subclass of Fl_Input that displays its input as a string of placeholders.

Depending on the platform this placeholder is either the asterisk ('∗') or the Unicode bullet character (U+2022).

This subclass is usually used to receive passwords and other "secret" information.

31.121.2 Constructor & Destructor Documentation

Fl_Secret_Input::Fl_Secret_Input (int *X*, int *Y*, int *W*, int *H*, const char ∗ *l* = 0)

Creates a new Fl_Secret_Input widget using the given position, size, and label string.

The default boxtype is FL_DOWN_BOX.

Inherited destructor destroys the widget and any value associated with it.

31.121.3 Member Function Documentation

int Fl_Secret_Input::handle (int *event*) [virtual]

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited handle() method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters

in	*event*	the kind of event received

Return values

0	if the event was not used or understood
1	if the event was used and can be deleted

See Also

Fl_Event

Reimplemented from Fl_Input.

The documentation for this class was generated from the following files:

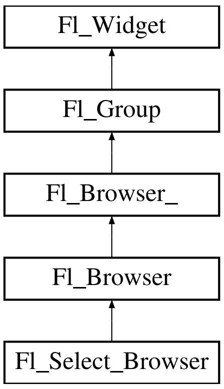
- Fl_Secret_Input.H
- Fl_Input.cxx

31.122 Fl_Select_Browser Class Reference

The class is a subclass of Fl_Browser which lets the user select a single item, or no items by clicking on the empty space.

`#include <Fl_Select_Browser.H>`

Inheritance diagram for Fl_Select_Browser:



Public Member Functions

- Fl_Select_Browser (int X, int Y, int W, int H, const char ∗L=0)

  Creates a new Fl_Select_Browser widget using the given position, size, and label string.

Additional Inherited Members

31.122.1 Detailed Description

The class is a subclass of Fl_Browser which lets the user select a single item, or no items by clicking on the empty space.

As long as the mouse button is held down on an unselected item it is highlighted. Normally the callback is done when the user presses the mouse, but you can change this with when().

See Fl_Browser for methods to add and remove lines from the browser.

31.122.2 Constructor & Destructor Documentation

Fl_Select_Browser::Fl_Select_Browser (int *X*, int *Y*, int *W*, int *H*, const char ∗ *L* = 0)

Creates a new Fl_Select_Browser widget using the given position, size, and label string.

The default boxtype is FL_DOWN_BOX. The constructor specializes Fl_Browser() by setting the type to FL_SELECT_BROWSER. The destructor destroys the widget and frees all memory that has been allocated.

The documentation for this class was generated from the following files:

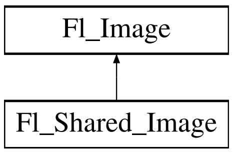
- Fl_Select_Browser.H
- Fl_Browser.cxx

31.123 Fl_Shared_Image Class Reference

This class supports caching, loading, scaling, and drawing of image files.

`#include <Fl_Shared_Image.H>`

Inheritance diagram for Fl_Shared_Image:



Public Member Functions

- virtual void color_average (Fl_Color c, float i)

  The color_average() method averages the colors in the image with the FLTK color value c.
- virtual Fl_Image ∗ copy (int W, int H)

  The copy() method creates a copy of the specified image.
- Fl_Image ∗ **copy** ()
- virtual void desaturate ()

  The desaturate() method converts an image to grayscale.
- virtual void draw (int X, int Y, int W, int H, int cx, int cy)

  Draws the image with a bounding box.
- void **draw** (int X, int Y)
- const char ∗ name ()

  Returns the filename of the shared image.
- int refcount ()

  Returns the number of references of this shared image.
- void release ()

  Releases and possibly destroys (if refcount <= 0) a shared image.
- void reload ()

  Reloads the shared image from disk.
- void scale (int width, int height, int proportional=1, int can_expand=0)

  Sets the drawing size of the shared image.
- virtual void uncache ()

  If the image has been cached for display, delete the cache data.

Static Public Member Functions

- static void add_handler (Fl_Shared_Handler f)

  Adds a shared image handler, which is basically a test function for adding new formats.
- static Fl_Shared_Image ∗ find (const char ∗name, int W=0, int H=0)

  Finds a shared image from its name and size specifications.
- static Fl_Shared_Image ∗ get (const char ∗name, int W=0, int H=0)

  Find or load an image that can be shared by multiple widgets.
- static Fl_Shared_Image ∗ get (Fl_RGB_Image ∗rgb, int own_it=1)

  Builds a shared image from a pre-existing Fl_RGB_Image.
- static Fl_Shared_Image ∗∗ images ()

  Returns the Fl_Shared_Image∗ array.
- static int num_images ()

  Returns the total number of shared images in the array.
- static void remove_handler (Fl_Shared_Handler f)

  Removes a shared image handler.
- static void scaling_algorithm (Fl_RGB_Scaling algorithm)

  Sets what algorithm is used when resizing a source image.

Protected Member Functions

- void add ()

  Adds a shared image to the image cache.
- Fl_Shared_Image ()

  Creates an empty shared image.
- Fl_Shared_Image (const char ∗n, Fl_Image ∗img=0)

  Creates a shared image from its filename and its corresponding Fl_Image∗ img.
- void **update** ()
- virtual ∼Fl_Shared_Image ()

  The destructor frees all memory and server resources that are used by the image.

Static Protected Member Functions

- static int compare (Fl_Shared_Image ∗∗i0, Fl_Shared_Image ∗∗i1)

  Compares two shared images.

Protected Attributes

- int **alloc_image_**
- Fl_Image ∗ **image_**
- const char ∗ **name_**
- int **original_**
- int **refcount_**

Static Protected Attributes

- static int **alloc_handlers_** = 0
- static int **alloc_images_** = 0
- static Fl_Shared_Handler ∗ **handlers_** = 0
- static Fl_Shared_Image ∗∗ **images_** = 0
- static int **num_handlers_** = 0
- static int **num_images_** = 0

Friends

- class **Fl_JPEG_Image**
- class **Fl_PNG_Image**

Additional Inherited Members

31.123.1 Detailed Description

This class supports caching, loading, scaling, and drawing of image files.

Most applications will also want to link against the fltk_images library and call the fl_register_images() function to support standard image formats such as BMP, GIF, JPEG, and PNG.

Images can be requested (loaded) with Fl_Shared_Image::get(), find(), and some other methods. All images are cached in an internal list of shared images and should be released when they are no longer needed. A refcount is used to determine if a released image is to be destroyed with delete.

See Also

Fl_Shared_Image::get()
Fl_Shared_Image::find()
Fl_Shared_Image::release()

31.123.2 Constructor & Destructor Documentation

Fl_Shared_Image::Fl_Shared_Image ( ) `[protected]`

Creates an empty shared image.

The constructors create a new shared image record in the image cache.

The constructors are protected and cannot be used directly from a program. Use the get() method instead.

Fl_Shared_Image::Fl_Shared_Image (const char ∗ *n*, Fl_Image ∗ *img* = *0*) `[protected]`

Creates a shared image from its filename and its corresponding Fl_Image∗ img.

The constructors create a new shared image record in the image cache.

The constructors are protected and cannot be used directly from a program. Use the get() method instead.

Fl_Shared_Image::∼Fl_Shared_Image ( ) `[protected]`, `[virtual]`

The destructor frees all memory and server resources that are used by the image.

The destructor is protected and cannot be used directly from a program. Use the Fl_Shared_Image::release() method instead.

31.123.3 Member Function Documentation

void Fl_Shared_Image::add ( ) `[protected]`

Adds a shared image to the image cache.

This **protected** method adds an image to the cache, an ordered list of shared images. The cache is searched for a matching image whenever one is requested, for instance with Fl_Shared_Image::get() or Fl_Shared_Image::find().

void Fl_Shared_Image::color_average (Fl_Color *c*, float *i*) `[virtual]`

The color_average() method averages the colors in the image with the FLTK color value c.

The i argument specifies the amount of the original image to combine with the color, so a value of 1.0 results in no color blend, and a value of 0.0 results in a constant image of the specified color.

An internal copy is made of the original image before changes are applied, to avoid modifying the original image.

Reimplemented from Fl_Image.

int Fl_Shared_Image::compare (Fl_Shared_Image ∗∗ *i0*, Fl_Shared_Image ∗∗ *i1*) `[static]`, `[protected]`

Compares two shared images.

The order of comparison is:

1. Image name, usually the filename used to load it
2. Image width
3. Image height

A special case is considered if the width of one of the images is zero and the other image is marked `original`. In this case the images match, i.e. the comparison returns success (0).

An image is marked `original` if it was directly loaded from a file or from memory as opposed to copied and resized images.

This comparison is used in Fl_Shared_Image::find() to find an image that matches the requested one or to find the position where a new image should be entered into the sorted list of shared images.

It is usually used in two steps:

1. search with exact width and height
2. if not found, search again with width = 0 (and height = 0)

The first step will only return a match if the image exists with the same width and height. The second step will match if there is an image marked `original` with the same name, regardless of width and height.

Returns

Whether the images match or their relative sort order (see text).

Return values

0	the images match
<0	Image `i0` is *less* than image `i1`
>0	Image `i0` is *greater* than image `i1`

Fl_Image ∗ Fl_Shared_Image::copy (int *W*, int *H*) `[virtual]`

The copy() method creates a copy of the specified image.

If the width and height are provided, the image is resized to the specified size. The image should be deleted (or in the case of Fl_Shared_Image, released) when you are done with it.

Reimplemented from Fl_Image.

void Fl_Shared_Image::desaturate ( ) `[virtual]`

The desaturate() method converts an image to grayscale.

If the image contains an alpha channel (depth = 4), the alpha channel is preserved.

An internal copy is made of the original image before changes are applied, to avoid modifying the original image.

Reimplemented from Fl_Image.

void Fl_Shared_Image::draw (int *X*, int *Y*, int *W*, int *H*, int *cx*, int *cy*) `[virtual]`

Draws the image with a bounding box.

Arguments `X,Y,W,H` specify a bounding box for the image, with the origin (upper-left corner) of the image offset by the `cx` and `cy` arguments.

In other words: `fl_push_clip(X,Y,W,H)` is applied, the image is drawn with its upper-left corner at `X-cx,Y-cy` and its own width and height, `fl_pop_clip()` is applied.

Reimplemented from Fl_Image.

Fl_Shared_Image ∗ Fl_Shared_Image::find (const char ∗ *name*, int *W* = *0*, int *H* = *0*) `[static]`

Finds a shared image from its name and size specifications.

This uses a binary search in the image cache.

If the image `name` exists with the exact width `W` and height `H`, then it is returned.

If `W` == 0 and the image `name` exists with another size, then the **original** image with that `name` is returned.

In either case the refcount of the returned image is increased. The found image should be released with Fl_Shared_Image::release() when no longer needed.

Fl_Shared_Image ∗ Fl_Shared_Image::get (const char ∗ *name*, int *W* = *0*, int *H* = *0*) `[static]`

Find or load an image that can be shared by multiple widgets.

If the image exists with the requested size, this image will be returned.

If the image exists, but only with another size, then a new copy with the requested size (width `W` and height `H`) will be created as a resized copy of the original image. The new image is added to the internal list of shared images.

If the image does not yet exist, then a new image of the proper dimension is created from the filename `name`. The original image from filename `name` is always added to the list of shared images in its original size. If the requested size differs, then the resized copy with width `W` and height `H` is also added to the list of shared images.

Note

> If the sizes differ, then *two* images are created as mentioned above. This is intentional so the original image is cached and preserved. If you request the same image with another size later, then the **original** image will be found, copied, resized, and returned.

Shared JPEG and PNG images can also be created from memory by using their named memory access constructor.

You should release() the image when you're done with it.

Parameters

name	name of the image
W,H	desired size

See Also

> Fl_Shared_Image::find(const char ∗name, int W, int H)
> Fl_Shared_Image::release()
> Fl_JPEG_Image::Fl_JPEG_Image(const char ∗name, const unsigned char ∗data)
> Fl_PNG_Image::Fl_PNG_Image (const char ∗name_png, const unsigned char ∗buffer, int maxsize)

Fl_Shared_Image ∗ Fl_Shared_Image::get (Fl_RGB_Image ∗ *rgb*, int *own_it* = *1*) `[static]`

Builds a shared image from a pre-existing Fl_RGB_Image.

Parameters

`in`	*rgb*	an Fl_RGB_Image used to build a new shared image.
`in`	*own_it*	1 if the shared image should delete `rgb` when it is itself deleted, 0 otherwise

Version

> 1.3.4

int Fl_Shared_Image::num_images ( ) `[static]`

Returns the total number of shared images in the array.

int Fl_Shared_Image::refcount ( ) `[inline]`

Returns the number of references of this shared image.

When reference is below 1, the image is deleted.

void Fl_Shared_Image::release ( )

Releases and possibly destroys (if refcount <= 0) a shared image.

In the latter case, it will reorganize the shared image array so that no hole will occur.

void Fl_Shared_Image::reload ( )

Reloads the shared image from disk.

void Fl_Shared_Image::remove_handler (Fl_Shared_Handler *f*) `[static]`

Removes a shared image handler.

void Fl_Shared_Image::scale (int *width*, int *height*, int *proportional* = `1`, int *can_expand* = `0`)

Sets the drawing size of the shared image.

This function gives the shared image its own size, independently from the size of the original image that is typically larger. This can be useful to draw a shared image on a drawing surface whose resolution is higher than the drawing unit for this surface: all pixels of the original image become available to fill an area of the drawing surface sized at `width,height`. Examples of such drawing surfaces: laser printers, PostScript files, PDF printers, retina displays on Apple hardware.

Parameters

width,height	maximum width and height (in drawing units) to use when drawing the shared image
proportional	if not null, keep the width and height of the shared image proportional to those of its original image
can_expand	if null, the width and height of the shared image will not exceed those of the original image

Version

1.3.4 and requires compiling with FLTK_ABI_VERSION = 10304

Example code: scale an image to fit in a box

```
Fl_Box *b = ...  // a box
Fl_Shared_Image *shared = Fl_Shared_Image::get("/path/to/picture.jpeg");
      // read a picture file
shared->scale(b->w(), b->h(), 1); // set the drawing size of the shared image to the size of the box
b->image(shared); // use the shared image as the box image
b->align(FL_ALIGN_INSIDE | FL_ALIGN_CENTER |
     FL_ALIGN_CLIP); // the image is to be drawn centered in the box
```

static void Fl_Shared_Image::scaling_algorithm (Fl_RGB_Scaling *algorithm*) `[inline]`, `[static]`

Sets what algorithm is used when resizing a source image.

The default algorithm is FL_RGB_SCALING_BILINEAR. Drawing an Fl_Shared_Image is sometimes performed by first resizing the source image and then drawing the resized copy. This occurs, e.g., when drawing to screen under Linux or MSWindows after having called Fl_Shared_Image::scale(). This function controls what method is used when the image to be resized is an Fl_RGB_Image.

Version

1.3.4 and requires compiling with FLTK_ABI_VERSION = 10304

void Fl_Shared_Image::uncache ( ) `[virtual]`

If the image has been cached for display, delete the cache data.

This allows you to change the data used for the image and then redraw it without recreating an image object.

Reimplemented from Fl_Image.

The documentation for this class was generated from the following files:

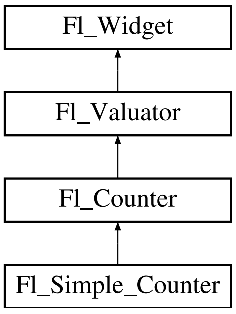
- Fl_Shared_Image.H
- Fl_Shared_Image.cxx

31.124 Fl_Simple_Counter Class Reference

This widget creates a counter with only 2 arrow buttons.

`#include <Fl_Simple_Counter.H>`

Inheritance diagram for Fl_Simple_Counter:



Public Member Functions

- **Fl_Simple_Counter** (int X, int Y, int W, int H, const char ∗L=0)

Additional Inherited Members

31.124.1 Detailed Description

This widget creates a counter with only 2 arrow buttons.

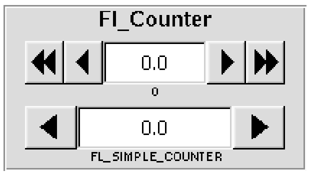


Figure 31.31: Fl_Simple_Counter

The documentation for this class was generated from the following files:

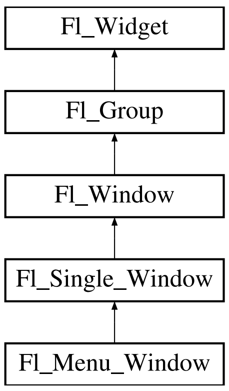
- Fl_Simple_Counter.H
- Fl_Counter.cxx

31.125 Fl_Single_Window Class Reference

This is the same as Fl_Window.

`#include <Fl_Single_Window.H>`

Inheritance diagram for Fl_Single_Window:



Public Member Functions

- Fl_Single_Window (int W, int H, const char ∗l=0)

  Creates a new Fl_Single_Window widget using the given size, and label (title) string.
- Fl_Single_Window (int X, int Y, int W, int H, const char ∗l=0)

  Creates a new Fl_Single_Window widget using the given position, size, and label (title) string.
- void flush ()

  Forces the window to be drawn, this window is also made current and calls draw().
- int **make_current** ()
- void show ()

  Puts the window on the screen.
- void **show** (int a, char ∗∗b)

Additional Inherited Members

31.125.1 Detailed Description

This is the same as Fl_Window.

However, it is possible that some implementations will provide double-buffered windows by default. This subclass can be used to force single-buffering. This may be useful for modifying existing programs that use incremental update, or for some types of image data, such as a movie flipbook.

31.125.2 Member Function Documentation

void Fl_Single_Window::flush ( ) `[virtual]`

Forces the window to be drawn, this window is also made current and calls draw().

Reimplemented from Fl_Window.

void Fl_Single_Window::show ( ) [virtual]

Puts the window on the screen.

Usually (on X) this has the side effect of opening the display.

If the window is already shown then it is restored and raised to the top. This is really convenient because your program can call show() at any time, even if the window is already up. It also means that show() serves the purpose of raise() in other toolkits.

Fl_Window::show(int argc, char **argv) is used for top-level windows and allows standard arguments to be parsed from the command-line.

Note

> For some obscure reasons Fl_Window::show() resets the current group by calling Fl_Group::current(0). The comments in the code say "get rid of very common user bug: forgot end()". Although this is true it may have unwanted side effects if you show() an unrelated window (maybe for an error message or warning) while building a window or any other group widget.

Todo Check if we can remove resetting the current group in a later FLTK version (after 1.3.x). This may break "already broken" programs though if they rely on this "feature".

See Also

> Fl_Window::show(int argc, char **argv)

Reimplemented from Fl_Window.

The documentation for this class was generated from the following files:

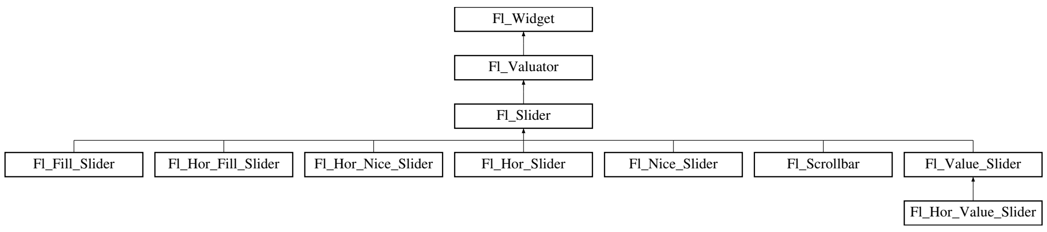
- Fl_Single_Window.H
- Fl_Single_Window.cxx

31.126 Fl_Slider Class Reference

The Fl_Slider widget contains a sliding knob inside a box.

`#include <Fl_Slider.H>`

Inheritance diagram for Fl_Slider:



Public Member Functions

- void bounds (double a, double b)

  Sets the minimum (a) and maximum (b) values for the valuator widget.
- Fl_Slider (int X, int Y, int W, int H, const char *L=0)

  Creates a new Fl_Slider widget using the given position, size, and label string.
- Fl_Slider (uchar t, int X, int Y, int W, int H, const char *L)

  Creates a new Fl_Slider widget using the given type, position, size, and label string.
- int handle (int)

Handles the specified event.

- int scrollvalue (int pos, int size, int first, int total)

  Sets the size and position of the sliding knob in the box.
- Fl_Boxtype slider () const

  Gets the slider box type.
- void slider (Fl_Boxtype c)

  Sets the slider box type.
- float slider_size () const

  Get the dimensions of the moving piece of slider.
- void slider_size (double v)

  Set the dimensions of the moving piece of slider.

Protected Member Functions

- void **draw** (int, int, int, int)
- void draw ()

  Draws the widget.
- int **handle** (int, int, int, int, int)

Additional Inherited Members

31.126.1 Detailed Description

The Fl_Slider widget contains a sliding knob inside a box.

It is often used as a scrollbar. Moving the box all the way to the top/left sets it to the minimum(), and to the bottom/right to the maximum(). The minimum() may be greater than the maximum() to reverse the slider direction.

Use void Fl_Widget::type(int) to set how the slider is drawn, which can be one of the following:

- FL_VERTICAL - Draws a vertical slider (this is the default).
- FL_HORIZONTAL - Draws a horizontal slider.
- FL_VERT_FILL_SLIDER - Draws a filled vertical slider, useful as a progress or value meter.
- FL_HOR_FILL_SLIDER - Draws a filled horizontal slider, useful as a progress or value meter.
- FL_VERT_NICE_SLIDER - Draws a vertical slider with a nice looking control knob.
- FL_HOR_NICE_SLIDER - Draws a horizontal slider with a nice looking control knob.

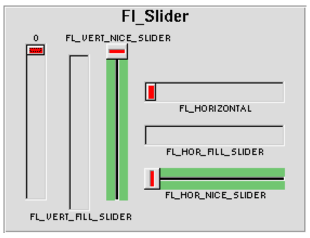


Figure 31.32: Fl_Slider

31.126.2 Constructor & Destructor Documentation

Fl_Slider::Fl_Slider (int *X*, int *Y*, int *W*, int *H*, const char ∗ *L* = 0)

Creates a new Fl_Slider widget using the given position, size, and label string.
The default boxtype is FL_DOWN_BOX.

31.126.3 Member Function Documentation

void Fl_Slider::bounds (double *a*, double *b*)

Sets the minimum (a) and maximum (b) values for the valuator widget.
if at least one of the values is changed, a partial redraw is asked.

void Fl_Slider::draw ( ) [protected], [virtual]

Draws the widget.
Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call redraw() instead.
Override this function to draw your own widgets.
If you ever need to call another widget's draw method *from within your own draw() method*, e.g. for an embedded scrollbar, you can do it (because draw() is virtual) like this:

```
Fl_Widget *s = &scroll;                 // scroll is an embedded Fl_Scrollbar
s->draw();                    // calls Fl_Scrollbar::draw()
```

Implements Fl_Widget.
Reimplemented in Fl_Value_Slider.

int Fl_Slider::handle (int *event*) [virtual]

Handles the specified event.
You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.
When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.
Most of the time, you want to call the inherited handle() method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.
Parameters

in	*event*	the kind of event received

Return values

0	if the event was not used or understood
1	if the event was used and can be deleted

See Also

Fl_Event

Reimplemented from Fl_Widget.
Reimplemented in Fl_Value_Slider.

int Fl_Slider::scrollvalue (int *pos*, int *size*, int *first*, int *total*)

Sets the size and position of the sliding knob in the box.

Parameters

in	*pos*	position of first line displayed
in	*size*	size of window in lines
in	*first*	number of first line
in	*total*	total number of lines Returns Fl_Valuator::value(p)

Fl_Boxtype Fl_Slider::slider ( ) const `[inline]`

Gets the slider box type.

void Fl_Slider::slider (Fl_Boxtype *c*) `[inline]`

Sets the slider box type.

void Fl_Slider::slider_size (double *v*)

Set the dimensions of the moving piece of slider.

This is the fraction of the size of the entire widget. If you set this to 1 then the slider cannot move. The default value is .08.

For the "fill" sliders this is the size of the area around the end that causes a drag effect rather than causing the slider to jump to the mouse.

The documentation for this class was generated from the following files:

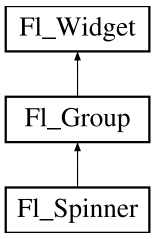
- Fl_Slider.H
- Fl_Slider.cxx

31.127 Fl_Spinner Class Reference

This widget is a combination of the input widget and repeat buttons.

`#include <Fl_Spinner.H>`

Inheritance diagram for Fl_Spinner:



Public Member Functions

- void color (Fl_Color v)

  Change the background color of the spinner widget's input field.
- Fl_Color color () const

  Return the background color of the spinner widget's input field.
- Fl_Spinner (int X, int Y, int W, int H, const char *L=0)

  Creates a new Fl_Spinner widget using the given position, size, and label string.
- const char * format ()

  Sets or returns the format string for the value.

- void format (const char ∗f)

  Sets or returns the format string for the value.
- int handle (int event)

  Handles the specified event.
- double maximum () const

  Gets the maximum value of the widget.
- void maximum (double m)

  Sets the maximum value of the widget.
- double maxinum () const

  Speling mistakes retained for source compatibility.
- double minimum () const

  Gets the minimum value of the widget.
- void minimum (double m)

  Sets the minimum value of the widget.
- double mininum () const

  Speling mistakes retained for source compatibility.
- void range (double a, double b)

  Sets the minimum and maximum values for the widget.
- void resize (int X, int Y, int W, int H)

  Resizes the Fl_Group widget and all of its children.
- void selection_color (Fl_Color val)

  Change the selection color of the spinner widget's input field.
- Fl_Color selection_color () const

  Return the selection color of the spinner widget's input field.
- double step () const

  Sets or returns the amount to change the value when the user clicks a button.
- void step (double s)

  See double Fl_Spinner::step() const.
- Fl_Color textcolor () const

  Gets the color of the text in the input field.
- void textcolor (Fl_Color c)

  Sets the color of the text in the input field.
- Fl_Font textfont () const

  Gets the font of the text in the input field.
- void textfont (Fl_Font f)

  Sets the font of the text in the input field.
- Fl_Fontsize textsize () const

  Gets the size of the text in the input field.
- void textsize (Fl_Fontsize s)

  Sets the size of the text in the input field.
- uchar type () const

  Gets the numeric representation in the input field.
- void type (uchar v)

  Sets the numeric representation in the input field.
- double value () const

  Gets the current value of the widget.
- void value (double v)

  Sets the current value of the widget.

Protected Attributes

- Fl_Repeat_Button **down_button_**
- Fl_Input **input_**
- Fl_Repeat_Button **up_button_**

Additional Inherited Members

31.127.1 Detailed Description

This widget is a combination of the input widget and repeat buttons.

The user can either type into the input area or use the buttons to change the value.

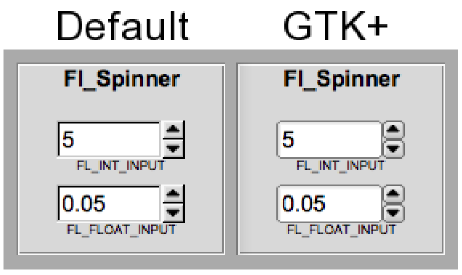


Figure 31.33: Fl_Spinner widget

31.127.2 Constructor & Destructor Documentation

Fl_Spinner::Fl_Spinner (int *X*, int *Y*, int *W*, int *H*, const char ∗ *L* = *0*)

Creates a new Fl_Spinner widget using the given position, size, and label string.

Inherited destructor Destroys the widget and any value associated with it.

31.127.3 Member Function Documentation

const char∗ Fl_Spinner::format ( ) `[inline]`

Sets or returns the format string for the value.

void Fl_Spinner::format (const char ∗ *f*) `[inline]`

Sets or returns the format string for the value.

int Fl_Spinner::handle (int *event*) `[inline]`, `[virtual]`

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited handle() method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters

in	*event*	the kind of event received

Return values

0	if the event was not used or understood
1	if the event was used and can be deleted

See Also

Fl_Event

Reimplemented from Fl_Group.

double Fl_Spinner::maximum ( ) const `[inline]`

Gets the maximum value of the widget.

void Fl_Spinner::maximum (double *m*) `[inline]`

Sets the maximum value of the widget.

double Fl_Spinner::maxinum ( ) const `[inline]`

Speling mistakes retained for source compatibility.

Deprecated

double Fl_Spinner::minimum ( ) const `[inline]`

Gets the minimum value of the widget.

void Fl_Spinner::minimum (double *m*) `[inline]`

Sets the minimum value of the widget.

double Fl_Spinner::mininum ( ) const `[inline]`

Speling mistakes retained for source compatibility.

Deprecated

void Fl_Spinner::range (double *a*, double *b*) `[inline]`

Sets the minimum and maximum values for the widget.

void Fl_Spinner::resize (int *X*, int *Y*, int *W*, int *H*) `[inline]`, `[virtual]`

Resizes the Fl_Group widget and all of its children.

The Fl_Group widget first resizes itself, and then it moves and resizes all its children according to the rules documented for Fl_Group::resizable(Fl_Widget∗)

See Also

Fl_Group::resizable(Fl_Widget∗)
Fl_Group::resizable()
Fl_Widget::resize(int,int,int,int)

Reimplemented from Fl_Group.

double Fl_Spinner::step ( ) const `[inline]`

Sets or returns the amount to change the value when the user clicks a button.

Before setting step to a non-integer value, the spinner type() should be changed to floating point.

Fl_Color Fl_Spinner::textcolor ( ) const `[inline]`

Gets the color of the text in the input field.

void Fl_Spinner::textcolor (Fl_Color ***c*** **)** `[inline]`

Sets the color of the text in the input field.

Fl_Font Fl_Spinner::textfont ( ) const `[inline]`

Gets the font of the text in the input field.

void Fl_Spinner::textfont (Fl_Font ***f*** **)** `[inline]`

Sets the font of the text in the input field.

Fl_Fontsize Fl_Spinner::textsize ( ) const `[inline]`

Gets the size of the text in the input field.

void Fl_Spinner::textsize (Fl_Fontsize ***s*** **)** `[inline]`

Sets the size of the text in the input field.

uchar Fl_Spinner::type ( ) const `[inline]`

Gets the numeric representation in the input field.

See Also

Fl_Spinner::type(uchar)

void Fl_Spinner::type (uchar ***v*** **)** `[inline]`

Sets the numeric representation in the input field.

Valid values are FL_INT_INPUT and FL_FLOAT_INPUT. Also changes the format() template. Setting a new spinner type via a superclass pointer will not work.

Note

type is not a virtual function.

double Fl_Spinner::value ( ) const `[inline]`

Gets the current value of the widget.

void Fl_Spinner::value (double ***v*** **)** `[inline]`

Sets the current value of the widget.

Before setting value to a non-integer value, the spinner type() should be changed to floating point.

The documentation for this class was generated from the following files:

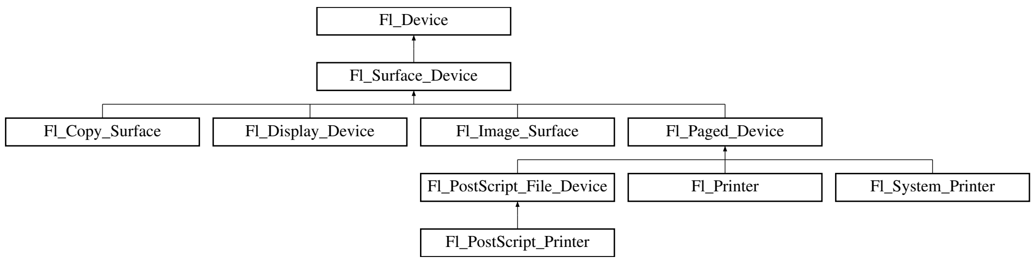
- Fl_Spinner.H
- Fl_Group.cxx

31.128 Fl_Surface_Device Class Reference

A drawing surface that's susceptible to receive graphical output.

`#include <Fl_Device.H>`

Inheritance diagram for Fl_Surface_Device:



Public Member Functions

- const char ∗ class_name ()

  Returns the name of the class of this object.
- void driver (Fl_Graphics_Driver ∗graphics_driver)

  Sets the graphics driver of this drawing surface.
- Fl_Graphics_Driver ∗ driver ()

  Returns the graphics driver of this drawing surface.
- virtual void set_current (void)

  Make this surface the current drawing surface.
- virtual ∼Fl_Surface_Device ()

  The destructor.

Static Public Member Functions

- static Fl_Surface_Device ∗ surface ()

  The current drawing surface.

Static Public Attributes

- static const char ∗ **class_id** = "Fl_Surface_Device"

Protected Member Functions

- Fl_Surface_Device (Fl_Graphics_Driver ∗graphics_driver)

  Constructor that sets the graphics driver to use for the created surface.

31.128.1 Detailed Description

A drawing surface that's susceptible to receive graphical output.

Any FLTK application has at any time a current drawing surface to which all drawing requests are directed. The current surface is given by Fl_Surface_Device::surface(). When main() begins running, the current drawing surface has been set to the computer's display, an instance of the Fl_Display_Device class.

A drawing surface other than the computer's display, is typically used as follows:

1. Create `surface`, an object from a particular Fl_Surface_Device derived class (e.g., Fl_Copy_-Surface, Fl_Printer).

2. Memorize what is the current drawing surface with `Fl_Surface_Device *old_current = Fl_Surface_Device::surface();`

3. Call `surface->set_current();` to redirect all graphics requests to `surface` which becomes the new current drawing surface (not necessary with class Fl_Printer because it is done by Fl_Printer::start_job()).

4. At this point any of the Drawing functions (e.g., fl_rect()) or the Color & Font functions or Drawing Images functions (e.g., fl_draw_image(), Fl_Image::draw()) operates on the new current drawing surface. Certain drawing surfaces allow additional ways to draw to them (e.g., Fl_Printer::print_widget(), Fl_Image_Surface::draw()).

5. After all drawing requests have been performed, redirect graphics requests back to their previous destination with `old_current->set_current();`.

6. Delete `surface`.

31.128.2 Constructor & Destructor Documentation

Fl_Surface_Device::Fl_Surface_Device (Fl_Graphics_Driver * *graphics_driver*) `[inline]`, `[protected]`

Constructor that sets the graphics driver to use for the created surface.

virtual Fl_Surface_Device::∼Fl_Surface_Device ( ) `[inline]`, `[virtual]`

The destructor.

31.128.3 Member Function Documentation

const char* Fl_Surface_Device::class_name ( ) `[inline]`, `[virtual]`

Returns the name of the class of this object.

Use of the class_name() function is discouraged because it will be removed from future FLTK versions.

The class of an instance of an Fl_Device subclass can be checked with code such as:

```
if ( instance->class_name() == Fl_Printer::class_id ) { ... }
```

Reimplemented from Fl_Device.

Reimplemented in Fl_Display_Device, Fl_PostScript_File_Device, Fl_Printer, Fl_Paged_Device, Fl_PostScript_Printer, Fl_System_Printer, and Fl_Image_Surface.

void Fl_Surface_Device::driver (Fl_Graphics_Driver * *graphics_driver*) `[inline]`

Sets the graphics driver of this drawing surface.

Fl_Graphics_Driver* Fl_Surface_Device::driver (void) `[inline]`

Returns the graphics driver of this drawing surface.

void Fl_Surface_Device::set_current (void) `[virtual]`

Make this surface the current drawing surface.

This surface will receive all future graphics requests.

Reimplemented in Fl_Printer, Fl_Copy_Surface, and Fl_Image_Surface.

static Fl_Surface_Device∗ Fl_Surface_Device::surface ( ) [inline], [static]

The current drawing surface.

In other words, the Fl_Surface_Device object that currently receives all graphics output

The documentation for this class was generated from the following files:

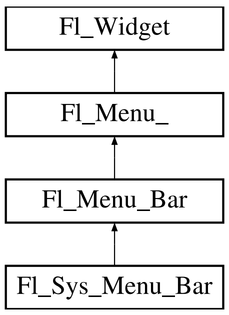
- Fl_Device.H
- Fl_Device.cxx

31.129 Fl_Sys_Menu_Bar Class Reference

A class to create, modify and delete menus that appear on Mac OS X in the menu bar at the top of the screen.

`#include <Fl_Sys_Menu_Bar.H>`

Inheritance diagram for Fl_Sys_Menu_Bar:



Public Member Functions

- int add (const char ∗label, int shortcut, Fl_Callback ∗, void ∗user_data=0, int flags=0)

  Add a new menu item to the system menu bar.
- int add (const char ∗label, const char ∗shortcut, Fl_Callback ∗cb, void ∗user_data=0, int flags=0)

  Adds a new menu item.
- int add (const char ∗str)

  Forms-compatible procedure to add items to the system menu bar.
- void clear ()

  Set the Fl_Menu_Item array pointer to null, indicating a zero-length menu.
- int clear_submenu (int index)

  Clears the specified submenu pointed to by index of all menu items.
- Fl_Sys_Menu_Bar (int x, int y, int w, int h, const char ∗l=0)

  The constructor.
- void global ()

  Make the shortcuts for this menu work no matter what window has the focus when you type it.
- int insert (int index, const char ∗label, int shortcut, Fl_Callback ∗cb, void ∗user_data=0, int flags=0)

  insert in the system menu bar a new menu item
- int insert (int index, const char ∗label, const char ∗shortcut, Fl_Callback ∗cb, void ∗user_data=0, int flags=0)

  Insert a new menu item.
- const Fl_Menu_Item ∗ menu () const

  Return the system menu's array of Fl_Menu_Item's.

- void menu (const Fl_Menu_Item ∗m)

  create a system menu bar using the given list of menu structs
- void mode (int i, int fl)

  Sets the flags of item i.
- int mode (int i) const

  Gets the flags of item i.
- void remove (int n)

  remove an item from the system menu bar
- void replace (int index, const char ∗name)

  rename an item from the system menu bar
- void setonly (Fl_Menu_Item ∗item)

  Turns the radio item "on" for the menu item and turns "off" adjacent radio items of the same group.
- void shortcut (int i, int s)

  Changes the shortcut of item i to n.
- void update ()

  Updates the system menu after any change to its items.
- ∼Fl_Sys_Menu_Bar ()

  The destructor.

Protected Member Functions

- void draw ()

  Draws the widget.

Additional Inherited Members

31.129.1 Detailed Description

A class to create, modify and delete menus that appear on Mac OS X in the menu bar at the top of the screen.

On other than Mac OS X platforms, Fl_Sys_Menu_Bar is a synonym of class Fl_Menu_Bar.

To use this class, just replace Fl_Menu_Bar by Fl_Sys_Menu_Bar, and, on the Mac platform, a system menu at the top of the screen will be available. This menu will match an array of Fl_Menu_Item's exactly as with standard FLTK menus.

Changes to the menu state are immediately visible in the menubar when they are made using member functions of the Fl_Sys_Menu_Bar class. Other changes (e.g., by a call to Fl_Menu_Item::set()) should be followed by a call to Fl_Sys_Menu_Bar::update() to be visible in the menubar across all platforms.

A few FLTK features are not supported by the Mac System menu:

- no symbolic labels
- no embossed labels
- no font sizes

You can configure a callback for the 'About' menu item to invoke your own code with fl_mac_set_about().

31.129.2 Constructor & Destructor Documentation

Fl_Sys_Menu_Bar::Fl_Sys_Menu_Bar (int *x*, int *y*, int *w*, int *h*, const char ∗ *l* = 0)

The constructor.

On Mac OS X, all arguments are unused. On other platforms they are used as by Fl_Menu_Bar::Fl_Menu_Bar().

31.129.3 Member Function Documentation

int Fl_Sys_Menu_Bar::add (const char * *label*, int *shortcut*, Fl_Callback * *cb*, void * *user_data* = *0*, int *flags* = *0*)

Add a new menu item to the system menu bar.

Parameters

label	- new menu item's label
shortcut	- new menu item's integer shortcut (can be 0 for none, or e.g. FL_ALT+'x')
cb	- callback to be invoked when item selected (can be 0 for none, in which case the menubar's callback() can be used instead)
user_data	- argument to the callback
flags	- item's flags, e.g. FL_MENU_TOGGLE, etc.

Returns

the index into the menu() array, where the entry was added

See Also

Fl_Menu_::add(const char∗ label, int shortcut, Fl_Callback ∗cb, void ∗user_data, int flags)

int Fl_Sys_Menu_Bar::add (const char ∗ *label*, const char ∗ *shortcut*, Fl_Callback ∗ *cb*, void ∗ *user_data* = *0*, int *flags* = *0*) `[inline]`

Adds a new menu item.

See Also

Fl_Menu_::add(const char∗ label, int shortcut, Fl_Callback∗, void ∗user_data=0, int flags=0)

int Fl_Sys_Menu_Bar::add (const char ∗ *str*)

Forms-compatible procedure to add items to the system menu bar.

Returns

the index into the menu() array, where the entry was added

See Also

Fl_Menu_::add(const char∗ str)

void Fl_Sys_Menu_Bar::clear ( )

Set the Fl_Menu_Item array pointer to null, indicating a zero-length menu.

See Also

Fl_Menu_::clear()

int Fl_Sys_Menu_Bar::clear_submenu (int *index*)

Clears the specified submenu pointed to by index of all menu items.

See Also

Fl_Menu_::clear_submenu(int index)

void Fl_Sys_Menu_Bar::draw ( ) `[protected]`, `[virtual]`

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call redraw() instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw() method*, e.g. for an embedded scrollbar, you can do it (because draw() is virtual) like this:

```
Fl_Widget *s = &scroll;               // scroll is an embedded Fl_Scrollbar
s->draw();                      // calls Fl_Scrollbar::draw()
```

Reimplemented from Fl_Menu_Bar.

int Fl_Sys_Menu_Bar::insert (int *index*, const char ∗ *label*, int *shortcut*, Fl_Callback ∗ *cb*, void ∗ *user_data* = 0, int *flags* = 0)

insert in the system menu bar a new menu item

Insert in the system menu bar a new menu item, with a title string, shortcut int, callback, argument to the callback, and flags.

Returns

the index into the menu() array, where the entry was inserted

See Also

Fl_Menu_::insert(int index, const char∗ label, int shortcut, Fl_Callback ∗cb, void ∗user_data, int flags)

int Fl_Sys_Menu_Bar::insert (int *index*, const char ∗ *label*, const char ∗ *shortcut*, Fl_Callback ∗ *cb*, void ∗ *user_data* = 0, int *flags* = 0) `[inline]`

Insert a new menu item.

See Also

Fl_Menu_::insert(int index, const char∗ label, const char∗ shortcut, Fl_Callback ∗cb, void ∗user_-data=0, int flags=0)

void Fl_Sys_Menu_Bar::menu (const Fl_Menu_Item ∗ *m*)

create a system menu bar using the given list of menu structs

Author

Matthias Melcher

Parameters

m	list of Fl_Menu_Item

void Fl_Sys_Menu_Bar::mode (int *i*, int *fl*) `[inline]`

Sets the flags of item i.

See Also

Fl_Menu_::mode(int i, int fl)

void Fl_Sys_Menu_Bar::remove (int *index*)

remove an item from the system menu bar

Parameters

index	the index of the item to remove

void Fl_Sys_Menu_Bar::replace (int *index*, const char ∗ *name*)

rename an item from the system menu bar

Parameters

index	the index of the item to rename
name	the new item name as a UTF8 string

void Fl_Sys_Menu_Bar::setonly (Fl_Menu_Item ∗ *item*) `[inline]`

Turns the radio item "on" for the menu item and turns "off" adjacent radio items of the same group.

The documentation for this class was generated from the following files:

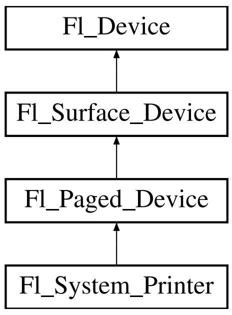
- Fl_Sys_Menu_Bar.H
- Fl_Sys_Menu_Bar.mm

31.130 Fl_System_Printer Class Reference

Print support under MSWindows and Mac OS.

`#include <Fl_Printer.H>`

Inheritance diagram for Fl_System_Printer:



Public Member Functions

- const char ∗ class_name ()

  Returns the name of the class of this object.
- void end_job (void)

  To be called at the end of a print job.
- int end_page (void)

  To be called at the end of each page.
- void margins (int ∗left, int ∗top, int ∗right, int ∗bottom)

  Computes the dimensions of margins that lie between the printable page area and the full page.
- void origin (int ∗x, int ∗y)

  Computes the page coordinates of the current origin of graphics functions.
- void origin (int x, int y)

  Sets the position in page coordinates of the origin of graphics functions.

- int printable_rect (int ∗w, int ∗h)

  Computes the width and height of the printable area of the page.
- void rotate (float angle)

  Rotates the graphics operations relatively to paper.
- void scale (float scale_x, float scale_y=0.)

  Changes the scaling of page coordinates.
- int start_job (int pagecount, int ∗frompage=NULL, int ∗topage=NULL)

  Starts a print job.
- int start_page (void)

  Starts a new printed page.
- void translate (int x, int y)

  Translates the current graphics origin accounting for the current rotation.
- void untranslate (void)

  Undoes the effect of a previous translate() call.
- ∼Fl_System_Printer (void)

  The destructor.

Static Public Attributes

- static const char ∗ **class_id** = Fl_Printer::class_id

Protected Member Functions

- Fl_System_Printer (void)

  The constructor.

Friends

- class **Fl_Printer**

Additional Inherited Members

31.130.1 Detailed Description

Print support under MSWindows and Mac OS.

Class Fl_System_Printer is implemented only on the MSWindows and Mac OS platforms. It has no public constructor. Use Fl_Printer instead that is cross-platform and has the same API.

31.130.2 Member Function Documentation

const char∗ Fl_System_Printer::class_name ( ) [inline], [virtual]

Returns the name of the class of this object.

Use of the class_name() function is discouraged because it will be removed from future FLTK versions.

The class of an instance of an Fl_Device subclass can be checked with code such as:

```
if ( instance->class_name() == Fl_Printer::class_id ) { ... }
```

Reimplemented from Fl_Paged_Device.

int Fl_System_Printer::end_page (void) `[virtual]`

To be called at the end of each page.

Returns

0 if OK, non-zero if any error.

Reimplemented from Fl_Paged_Device.

void Fl_System_Printer::margins (int * *left*, int * *top*, int * *right*, int * *bottom*) `[virtual]`

Computes the dimensions of margins that lie between the printable page area and the full page.

Values are in the same unit as that used by FLTK drawing functions. They are changed by scale() calls.

Parameters

`out`	*left*	If non-null, *left is set to the left margin size.
`out`	*top*	If non-null, *top is set to the top margin size.
`out`	*right*	If non-null, *right is set to the right margin size.
`out`	*bottom*	If non-null, *bottom is set to the bottom margin size.

Reimplemented from Fl_Paged_Device.

void Fl_System_Printer::origin (int * *x*, int * *y*) `[virtual]`

Computes the page coordinates of the current origin of graphics functions.

Parameters

`out`	*x*	If non-null, *x is set to the horizontal page offset of graphics origin.
`out`	*y*	Same as above, vertically.

Reimplemented from Fl_Paged_Device.

void Fl_System_Printer::origin (int *x*, int *y*) `[virtual]`

Sets the position in page coordinates of the origin of graphics functions.

Arguments should be expressed relatively to the result of a previous printable_rect() call. That is, `printable_rect(&w, &h); origin(w/2, 0);` sets the graphics origin at the top center of the page printable area. Origin() calls are not affected by rotate() calls. Successive origin() calls don't combine their effects.

Parameters

`in`	*x*	Horizontal position in page coordinates of the desired origin of graphics functions.
`in`	*y*	Same as above, vertically.

Reimplemented from Fl_Paged_Device.

int Fl_System_Printer::printable_rect (int * *w*, int * *h*) `[virtual]`

Computes the width and height of the printable area of the page.

Values are in the same unit as that used by FLTK drawing functions, are unchanged by calls to origin(), but are changed by scale() calls. Values account for the user-selected paper type and print orientation.

Returns

0 if OK, non-zero if any error

Reimplemented from Fl_Paged_Device.

void Fl_System_Printer::rotate (float *angle*) `[virtual]`

Rotates the graphics operations relatively to paper.

The rotation is centered on the current graphics origin. Successive rotate() calls don't combine their effects.

Parameters

angle	Rotation angle in counter-clockwise degrees.

Reimplemented from Fl_Paged_Device.

void Fl_System_Printer::scale (float *scale_x*, float *scale_y* = `0.`) `[virtual]`

Changes the scaling of page coordinates.

This function also resets the origin of graphics functions at top left of printable page area. After a scale() call, do a printable_rect() call to get the new dimensions of the printable page area. Successive scale() calls don't combine their effects.

Parameters

scale_x	Horizontal dimensions of plot are multiplied by this quantity.
scale_y	Same as above, vertically. The value 0. is equivalent to setting `scale_y` = `scale_x`. Thus, scale(factor); is equivalent to scale(factor, factor);

Reimplemented from Fl_Paged_Device.

int Fl_System_Printer::start_job (int *pagecount*, int ∗ *frompage* = `NULL`, int ∗ *topage* = `NULL`) `[virtual]`

Starts a print job.

Parameters

`in`	*pagecount*	the total number of pages of the job (or 0 if you don't know the number of pages)
`out`	*frompage*	if non-null, ∗frompage is set to the first page the user wants printed
`out`	*topage*	if non-null, ∗topage is set to the last page the user wants printed

Returns

0 if OK, non-zero if any error

Reimplemented from Fl_Paged_Device.

int Fl_System_Printer::start_page (void) `[virtual]`

Starts a new printed page.

The page coordinates are initially in points, i.e., 1/72 inch, and with origin at the top left of the printable page area.

Returns

0 if OK, non-zero if any error

Reimplemented from Fl_Paged_Device.

void Fl_System_Printer::translate (int *x*, int *y*) `[virtual]`

Translates the current graphics origin accounting for the current rotation.

This function is only useful after a rotate() call. Each translate() call must be matched by an untranslate() call. Successive translate() calls add up their effects.

Reimplemented from Fl_Paged_Device.

The documentation for this class was generated from the following files:

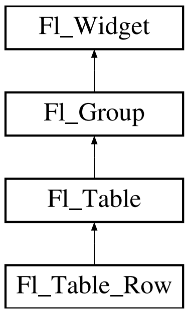
- Fl_Printer.H
- Fl_Printer.cxx

31.131 Fl_Table Class Reference

A table of widgets or other content.

`#include <Fl_Table.H>`

Inheritance diagram for Fl_Table:



Public Types

- enum TableContext {
  CONTEXT_NONE = 0, CONTEXT_STARTPAGE = 0x01, CONTEXT_ENDPAGE = 0x02, CONTEXT_ROW_HEADER = 0x04,
  CONTEXT_COL_HEADER = 0x08, CONTEXT_CELL = 0x10, CONTEXT_TABLE = 0x20, CONTEXT_RC_RESIZE = 0x40 }

  The context bit flags for Fl_Table related callbacks.

Public Member Functions

- void **add** (Fl_Widget &wgt)
- void **add** (Fl_Widget ∗wgt)
- Fl_Widget ∗const ∗ **array** ()
- void **begin** ()
- void callback (Fl_Widget ∗, void ∗)

  Callbacks will be called depending on the setting of Fl_Widget::when().
- int callback_col ()

  Returns the current column the event occurred on.
- TableContext callback_context ()

  Returns the current 'table context'.
- int callback_row ()

  Returns the current row the event occurred on.
- Fl_Widget ∗ child (int n) const

  Returns the child widget by an index.
- int children () const

  Returns the number of children in the table.
- virtual void clear ()

  Clears the table to zero rows (rows(0)), zero columns (cols(0)), and clears any widgets (table->clear()) that were added with begin()/end() or add()/insert()/etc.

- int col_header ()

  Returns if column headers are enabled or not.
- void col_header (int flag)

  Enable or disable column headers.
- void col_header_color (Fl_Color val)

  Sets the color for column headers and redraws the table.
- Fl_Color col_header_color ()

  Gets the color for column headers.
- void col_header_height (int height)

  Sets the height in pixels for column headers and redraws the table.
- int col_header_height ()

  Gets the column header height.
- void col_position (int col)

  Sets the column scroll position to column 'col', and causes the screen to redraw.
- int col_position ()

  Returns the current column scroll position as a column number.
- int col_resize ()

  Returns if column resizing by the user is allowed.
- void col_resize (int flag)

  Allows/disallows column resizing by the user.
- int col_resize_min ()

  Returns the current column minimum resize value.
- void col_resize_min (int val)

  Sets the current column minimum resize value.
- void col_width (int col, int width)

  Sets the width of the specified column in pixels, and the table is redrawn.
- int col_width (int col)

  Returns the current width of the specified column in pixels.
- void col_width_all (int width)

  Convenience method to set the width of all columns to the same value, in pixels.
- virtual void cols (int val)

  Set the number of columns in the table and redraw.
- int cols ()

  Get the number of columns in the table.
- void **do_callback** (TableContext context, int row, int col)
- void draw (void)

  Draws the widget.
- void **end** ()
- int **find** (const Fl_Widget ∗wgt) const
- int **find** (const Fl_Widget &wgt) const
- Fl_Table (int X, int Y, int W, int H, const char ∗l=0)

  The constructor for the Fl_Table.
- void get_selection (int &row_top, int &col_left, int &row_bot, int &col_right)

  Gets the region of cells selected (highlighted).
- void **init_sizes** ()
- void **insert** (Fl_Widget &wgt, int n)
- void **insert** (Fl_Widget &wgt, Fl_Widget ∗w2)

- int is_interactive_resize ()

  Returns 1 if someone is interactively resizing a row or column.
- int is_selected (int r, int c)

  See if the cell at row `r` and column `c` is selected.
- int **move_cursor** (int R, int C, int shiftselect)
- int **move_cursor** (int R, int C)
- void **remove** (Fl_Widget &wgt)
- void resize (int X, int Y, int W, int H)

  Changes the size of the Fl_Table, causing it to redraw.
- int row_header ()

  Returns if row headers are enabled or not.
- void row_header (int flag)

  Enables/disables showing the row headers.
- void row_header_color (Fl_Color val)

  Sets the row header color and causes the screen to redraw.
- Fl_Color row_header_color ()

  Returns the current row header color.
- void row_header_width (int width)

  Sets the row header width to n and causes the screen to redraw.
- int row_header_width ()

  Returns the current row header width (in pixels).
- void row_height (int row, int height)

  Sets the height of the specified row in pixels, and the table is redrawn.
- int row_height (int row)

  Returns the current height of the specified row as a value in pixels.
- void row_height_all (int height)

  Convenience method to set the height of all rows to the same value, in pixels.
- void row_position (int row)

  Sets the row scroll position to 'row', and causes the screen to redraw.
- int row_position ()

  Returns the current row scroll position as a row number.
- int row_resize ()

  Returns if row resizing by the user is allowed.
- void row_resize (int flag)

  Allows/disallows row resizing by the user.
- int row_resize_min ()

  Returns the current row minimum resize value.
- void row_resize_min (int val)

  Sets the current row minimum resize value.
- virtual void rows (int val)

  Sets the number of rows in the table, and the table is redrawn.
- int rows ()

  Returns the number of rows in the table.
- int scrollbar_size () const

  Gets the current size of the scrollbars' troughs, in pixels.
- void scrollbar_size (int newSize)

  Sets the pixel size of the scrollbars' troughs to `newSize`, in pixels.

- void set_selection (int row_top, int col_left, int row_bot, int col_right)

    Sets the region of cells to be selected (highlighted).
- void tab_cell_nav (int val)

    Flag to control if Tab navigates table cells or not.
- int tab_cell_nav () const

    Get state of table's 'Tab' key cell navigation flag.
- void table_box (Fl_Boxtype val)

    Sets the kind of box drawn around the data table, the default being FL_NO_BOX.
- Fl_Boxtype table_box (void)

    Returns the current box type used for the data table.
- void top_row (int row)

    Sets which row should be at the top of the table, scrolling as necessary, and the table is redrawn.
- int top_row ()

    Returns the current top row shown in the table.
- void visible_cells (int &r1, int &r2, int &c1, int &c2)

    Returns the range of row and column numbers for all visible and partially visible cells in the table.
- void when (Fl_When flags)

    The Fl_Widget::when() function is used to set a group of flags, determining when the widget callback is called:
- ~Fl_Table ()

    The destructor for the Fl_Table.

Protected Types

- enum **ResizeFlag** {
  RESIZE_NONE = 0, **RESIZE_COL_LEFT** = 1, **RESIZE_COL_RIGHT** = 2, **RESIZE_ROW_A-BOVE** = 3,
  RESIZE_ROW_BELOW = 4 }

Protected Member Functions

- void **change_cursor** (Fl_Cursor newcursor)
- long **col_scroll_position** (int col)
- TableContext **cursor2rowcol** (int &R, int &C, ResizeFlag &resizeflag)
- void **damage_zone** (int r1, int c1, int r2, int c2, int r3=0, int c3=0)
- virtual void draw_cell (TableContext context, int R=0, int C=0, int X=0, int Y=0, int W=0, int H=0)

    Subclass should override this method to handle drawing the cells.
- int **find_cell** (TableContext context, int R, int C, int &X, int &Y, int &W, int &H)
- void **get_bounds** (TableContext context, int &X, int &Y, int &W, int &H)
- int handle (int e)

    Handles the specified event.
- int **is_fltk_container** ()
- void **recalc_dimensions** ()
- void **redraw_range** (int topRow, int botRow, int leftCol, int rightCol)
- int **row_col_clamp** (TableContext context, int &R, int &C)
- long **row_scroll_position** (int row)
- void **table_resized** ()
- void **table_scrolled** ()

Static Protected Member Functions

- static void **scroll_cb** (Fl_Widget ∗, void ∗)

Protected Attributes

- int **botrow**
- int **current_col**
- int **current_row**
- Fl_Scrollbar ∗ **hscrollbar**
- int **leftcol**
- int **leftcol_scrollpos**
- int **rightcol**
- int **select_col**
- int **select_row**
- Fl_Scroll ∗ **table**
- int **table_h**
- int **table_w**
- int **tih**
- int **tiw**
- int **tix**
- int **tiy**
- int **toh**
- int **toprow**
- int **toprow_scrollpos**
- int **tow**
- int **tox**
- int **toy**
- Fl_Scrollbar ∗ **vscrollbar**
- int **wih**
- int **wiw**
- int **wix**
- int **wiy**

Additional Inherited Members

31.131.1 Detailed Description

A table of widgets or other content.

This is the base class for table widgets.

To be useful it must be subclassed and several virtual functions defined. Normally applications use widgets derived from this widget, and do not use this widget directly; this widget is usually too low level to be used directly by applications.

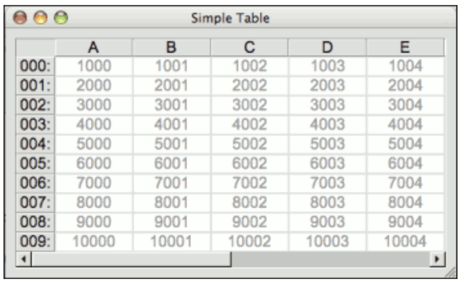
This widget does *not* handle the data in the table. The draw_cell() method must be overridden by a subclass to manage drawing the contents of the cells.

This widget can be used in several ways:

- As a custom widget; see examples/table-simple.cxx and test/table.cxx. Very optimal for even extremely large tables.

- As a table made up of a single FLTK widget instanced all over the table, simulating a numeric spreadsheet. See examples/table-spreadsheet.cxx and examples/table-spreadsheet-with-keyboard-nav.cxx. Optimal for large tables.

- As a regular container of FLTK widgets, one widget per cell. See examples/table-as-container.cxx. *Not* recommended for large tables.


Simple Table

	A	B	C	D	E
000:	1000	1001	1002	1003	1004
001:	2000	2001	2002	2003	2004
002:	3000	3001	3002	3003	3004
003:	4000	4001	4002	4003	4004
004:	5000	5001	5002	5003	5004
005:	6000	6001	6002	6003	6004
006:	7000	7001	7002	7003	7004
007:	8000	8001	8002	8003	8004
008:	9000	9001	9002	9003	9004
009:	10000	10001	10002	10003	10004

Figure 31.34: table-simple example

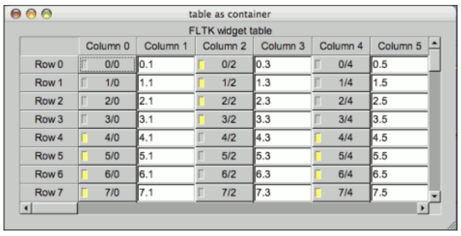


table as container

FLTK widget table

	Column 0	Column 1	Column 2	Column 3	Column 4	Column 5
Row 0	0/0	0.1	0/2	0.3	0/4	0.5
Row 1	1/0	1.1	1/2	1.3	1/4	1.5
Row 2	2/0	2.1	2/2	2.3	2/4	2.5
Row 3	3/0	3.1	3/2	3.3	3/4	3.5
Row 4	4/0	4.1	4/2	4.3	4/4	4.5
Row 5	5/0	5.1	5/2	5.3	5/4	5.5
Row 6	6/0	6.1	6/2	6.3	6/4	6.5
Row 7	7/0	7.1	7/2	7.3	7/4	7.5

Figure 31.35: table-as-container example

When acting as part of a custom widget, events on the cells and/or headings generate callbacks when they are clicked by the user. You control when events are generated based on the setting for Fl_Table::when().

When acting as a container for FLTK widgets, the FLTK widgets maintain themselves. Although the draw_cell() method must be overridden, its contents can be very simple. See the draw_cell() code in examples/table-simple.cxx.

The following variables are available to classes deriving from Fl_Table:

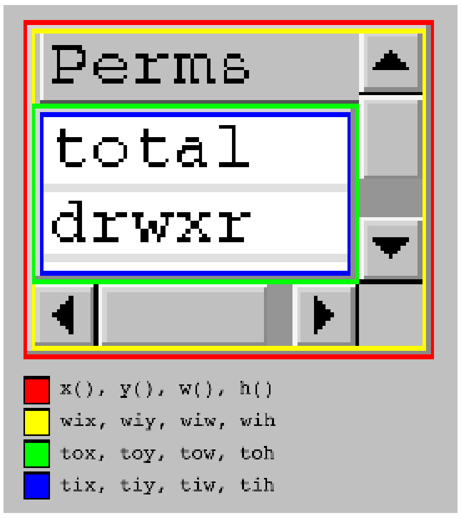


Figure 31.36: Fl_Table Dimensions

x()/y()/w()/h()	Fl_Table widget's outer dimension. The outer edge of the border of the Fl_Table. (Red in the diagram above)
wix/wiy/wiw/wih	Fl_Table widget's inner dimension. The inner edge of the border of the Fl_Table. eg. if the Fl_Table's box() is FL_NO_BOX, these values are the same as x()/y()/w()/h(). (Yellow in the diagram above)
tox/toy/tow/toh	The table's outer dimension. The outer edge of the border around the cells, but inside the row/col headings and scrollbars. (Green in the diagram above)
tix/tiy/tiw/tih	The table's inner dimension. The inner edge of the border around the cells, but inside the row/col headings and scrollbars. AKA the table's clip region. eg. if the table_box() is FL_NO_BOX, these values are the same as tox/toy/tow/toh. (Blue in the diagram above)

CORE DEVELOPERS

- Greg Ercolano : 12/16/2002 - initial implementation 12/16/02. Fl_Table, Fl_Table_Row, docs.
- Jean-Marc Lienher : 02/22/2004 - added keyboard nav + mouse selection, and ported Fl_Table into fltk-utf8-1.1.4

OTHER CONTRIBUTORS

- Inspired by the Feb 2000 version of FLVW's Flvw_Table widget. Mucho thanks to those folks.
- Mister Satan : 04/07/2003 - MinGW porting mods, and singleinput.cxx; a cool Fl_Input oriented spreadsheet example
- Marek Paliwoda : 01/08/2003 - Porting mods for Borland

- Ori Berger : 03/16/2006 - Optimizations for >500k rows/cols

LICENSE

Greg added the following license to the original distribution of Fl_Table. He kindly gave his permission to integrate Fl_Table and Fl_Table_Row into FLTK, allowing FLTK license to apply while his widgets are part of the library.

If used on its own, this is the license that applies:

```
Fl_Table License
December 16, 2002

The Fl_Table library and included programs are provided under the terms
of the GNU Library General Public License (LGPL) with the following
exceptions:

1. Modifications to the Fl_Table configure script, config
header file, and makefiles by themselves to support
a specific platform do not constitute a modified or
derivative work.

The authors do request that such modifications be
contributed to the Fl_Table project - send all
contributions to "erco at seriss dot com".

2. Widgets that are subclassed from Fl_Table widgets do not
constitute a derivative work.

3. Static linking of applications and widgets to the
Fl_Table library does not constitute a derivative work
and does not require the author to provide source
code for the application or widget, use the shared
Fl_Table libraries, or link their applications or
widgets against a user-supplied version of Fl_Table.

If you link the application or widget to a modified
version of Fl_Table, then the changes to Fl_Table must be
provided under the terms of the LGPL in sections
1, 2, and 4.

4. You do not have to provide a copy of the Fl_Table license
with programs that are linked to the Fl_Table library, nor
do you have to identify the Fl_Table license in your
program or documentation as required by section 6
of the LGPL.

However, programs must still identify their use of Fl_Table.
The following example statement can be included in user
documentation to satisfy this requirement:

[program/widget] is based in part on the work of
the Fl_Table project http://seriss.com/people/erco/fltk/Fl_Table/
```

31.131.2 Member Enumeration Documentation

enum Fl_Table::TableContext

The context bit flags for Fl_Table related callbacks.

Used in draw_cell(), callback(), etc.

Enumerator

CONTEXT_NONE no known context

CONTEXT_STARTPAGE before a page is redrawn

CONTEXT_ENDPAGE after a page is redrawn

CONTEXT_ROW_HEADER in the row header

CONTEXT_COL_HEADER in the col header

CONTEXT_CELL in one of the cells

CONTEXT_TABLE in a dead zone of table

CONTEXT_RC_RESIZE column or row being resized

31.131.3 Constructor & Destructor Documentation

Fl_Table::Fl_Table (int *X*, int *Y*, int *W*, int *H*, const char ∗ *l* = `0`)

The constructor for the Fl_Table.

This creates an empty table with no rows or columns, with headers and row/column resize behavior disabled.

Fl_Table::∼Fl_Table ( )

The destructor for the Fl_Table.

Destroys the table and its associated widgets.

31.131.4 Member Function Documentation

void Fl_Table::callback (Fl_Widget ∗ , void ∗)

Callbacks will be called depending on the setting of Fl_Widget::when().

Callback functions should use the following functions to determine the context/row/column:

- Fl_Table::callback_row() returns current row
- Fl_Table::callback_col() returns current column
- Fl_Table::callback_context() returns current table context

callback_row() and callback_col() will be set to the row and column number the event occurred on. If someone clicked on a row header, `col` will be *0*. If someone clicked on a column header, `row` will be *0*.

callback_context() will return one of the following:

`Fl_Table::CONTEXT_ROW_HEADER`	Someone clicked on a row header. Excludes resizing.
`Fl_Table::CONTEXT_COL_HEADER`	Someone clicked on a column header. Excludes resizing.
`Fl_Table::CONTEXT_CELL`	Someone clicked on a cell. To receive callbacks for FL_RELEASE events, you must set when(FL_WHEN_RELEASE).
`Fl_Table::CONTEXT_RC_RESIZE`	Someone is resizing rows/columns either interactively, or via the col_width() or row_height() API. Use is_interactive_resize() to determine interactive resizing. If resizing a column, R=0 and C=column being resized. If resizing a row, C=0 and R=row being resized. NOTE: To receive resize events, you must set when(FL_WHEN_CHANGED).

```
class MyTable : public Fl_Table {
  [..]
private:
  // Handle events that happen on the table
  void event_callback2() {
    int R = callback_row(),                         // row where event occurred
    C = callback_col();                             // column where event occurred
    TableContext context = callback_context();      // which part of table
    fprintf(stderr, "callback: Row=%d Col=%d Context=%d Event=%d\n",
            R, C, (int)context, (int)Fl::event());
  }

  // Actual static callback
  static void event_callback(Fl_Widget*, void* data) {
    MyTable *o = (MyTable*)data;
    o->event_callback2();
  }

public:
  // Constructor
  MyTable() {
    [..]
    table.callback(&event_callback, (void*)this);   // setup callback
    table.when(FL_WHEN_CHANGED|FL_WHEN_RELEASE);    // when to call it
  }
};
```

int Fl_Table::callback_col ( ) [inline]

Returns the current column the event occurred on.

This function should only be used from within the user's callback function.

TableContext Fl_Table::callback_context ( ) [inline]

Returns the current 'table context'.

This function should only be used from within the user's callback function.

int Fl_Table::callback_row ( ) [inline]

Returns the current row the event occurred on.

This function should only be used from within the user's callback function.

Fl_Widget* Fl_Table::child (int *n*) const [inline]

Returns the child widget by an index.

When using the Fl_Table as a container for FLTK widgets, this method returns the widget pointer from the internal array of widgets in the container.

Typically used in loops, eg:

```
for ( int i=0; i<children(); i++ ) {
  Fl_Widget *w = child(i);
  [..]
}
```

int Fl_Table::children ( ) const [inline]

Returns the number of children in the table.

When using the Fl_Table as a container for FLTK widgets, this method returns how many child widgets the table has.

See Also

child(int)

virtual void Fl_Table::clear ( ) [inline], [virtual]

Clears the table to zero rows (rows(0)), zero columns (cols(0)), and clears any widgets (table->clear()) that were added with begin()/end() or add()/insert()/etc.

See Also

rows(int), cols(int)

Reimplemented in Fl_Table_Row.

void Fl_Table::col_header (int *flag*) [inline]

Enable or disable column headers.

If changed, the table is redrawn.

void Fl_Table::col_resize (int *flag*) [inline]

Allows/disallows column resizing by the user.

1=allow interactive resizing, 0=disallow interactive resizing. Since interactive resizing is done via the column headers, `col_header()` must also be enabled to allow resizing.

void Fl_Table::col_resize_min (int *val*) [inline]

Sets the current column minimum resize value.

This is used to prevent the user from interactively resizing any column to be smaller than 'pixels'. Must be a value >=1.

void Fl_Table::col_width (int *col*, int *width*)

Sets the width of the specified column in pixels, and the table is redrawn.

callback() will be invoked with CONTEXT_RC_RESIZE if the column's width was actually changed, and when() is FL_WHEN_CHANGED.

void Fl_Table::col_width_all (int *width*) [inline]

Convenience method to set the width of all columns to the same value, in pixels.

The screen is redrawn.

void Fl_Table::draw (void) [virtual]

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call redraw() instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw() method*, e.g. for an embedded scrollbar, you can do it (because draw() is virtual) like this:

```
Fl_Widget *s = &scroll;              // scroll is an embedded Fl_Scrollbar
s->draw();                     // calls Fl_Scrollbar::draw()
```

Reimplemented from Fl_Group.

virtual void Fl_Table::draw_cell (TableContext *context*, int *R* = *0*, int *C* = *0*, int *X* = *0*, int *Y* = *0*, int *W* = *0*, int *H* = *0*) `[inline]`, `[protected]`, `[virtual]`

Subclass should override this method to handle drawing the cells.

This method will be called whenever the table is redrawn, once per cell.

Only cells that are completely (or partially) visible will be told to draw.

`context` will be one of the following:

`Fl_Table::CONTEXT_STARTPAGE`	When table, or parts of the table, are about to be redrawn. Use to initialize static data, such as font selections. R/C will be zero, X/Y/W/H will be the dimensions of the table's entire data area. (Useful for locking a database before accessing; see also visible_cells())
`Fl_Table::CONTEXT_ENDPAGE`	When table has completed being redrawn. R/C will be zero, X/Y/W/H dimensions of table's data area. (Useful for unlocking a database after accessing)
`Fl_Table::CONTEXT_ROW_HEADER`	Whenever a row header cell needs to be drawn. R will be the row number of the header being redrawn, C will be zero, X/Y/W/H will be the fltk drawing area of the row header in the window
`Fl_Table::CONTEXT_COL_HEADER`	Whenever a column header cell needs to be drawn. R will be zero, C will be the column number of the header being redrawn, X/Y/W/H will be the fltk drawing area of the column header in the window
`Fl_Table::CONTEXT_CELL`	Whenever a data cell in the table needs to be drawn. R/C will be the row/column of the cell to be drawn, X/Y/W/H will be the fltk drawing area of the cell in the window
`Fl_Table::CONTEXT_RC_RESIZE`	Whenever table or row/column is resized or scrolled, either interactively or via col_width() or row_height(). R/C/X/Y/W/H will all be zero. Useful for fltk containers that need to resize or move the child fltk widgets.

`row` and `col` will be set to the row and column number of the cell being drawn. In the case of row headers, `col` will be *0*. In the case of column headers, `row` will be *0*.

`x/y/w/h` will be the position and dimensions of where the cell should be drawn.

In the case of custom widgets, a minimal draw_cell() override might look like the following. With custom widgets it is up to the caller to handle drawing everything within the dimensions of the cell, including handling the selection color. Note all clipping must be handled as well; this allows drawing outside the dimensions of the cell if so desired for 'custom effects'.

```
// This is called whenever Fl_Table wants you to draw a cell
void MyTable::draw_cell(TableContext context, int R=0, int C=0, int X=0, int Y=0, int W=0, int
      H=0) {
    static char s[40];
    sprintf(s, "%d/%d", R, C);              // text for each cell
    switch ( context ) {
        case CONTEXT_STARTPAGE:             // Fl_Table telling us it's starting to draw
       page
            fl_font(FL_HELVETICA, 16);
            return;

        case CONTEXT_ROW_HEADER:            // Fl_Table telling us to draw row/col
       headers
        case CONTEXT_COL_HEADER:
            fl_push_clip(X, Y, W, H);
            {
           fl_draw_box(FL_THIN_UP_BOX, X, Y, W, H,
      color());
           fl_color(FL_BLACK);
           fl_draw(s, X, Y, W, H, FL_ALIGN_CENTER);
            }
            fl_pop_clip();
            return;

        case CONTEXT_CELL:                  // Fl_Table telling us to draw cells
            fl_push_clip(X, Y, W, H);
            {
           // BG COLOR
           fl_color( row_selected(R) ? selection_color() : FL_WHITE);
           fl_rectf(X, Y, W, H);

           // TEXT
           fl_color(FL_BLACK);
           fl_draw(s, X, Y, W, H, FL_ALIGN_CENTER);

           // BORDER
           fl_color(FL_LIGHT2);
           fl_rect(X, Y, W, H);
            }
            fl_pop_clip();
            return;

    default:
        return;
    }
    //NOTREACHED
}
```

void Fl_Table::get_selection (int & *row_top,* int & *col_left,* int & *row_bot,* int & *col_right*)

Gets the region of cells selected (highlighted).

Parameters

in	*row_top*	Returns the top row of selection area
in	*col_left*	Returns the left column of selection area
in	*row_bot*	Returns the bottom row of selection area
in	*col_right*	Returns the right column of selection area

int Fl_Table::handle (int *event*) `[protected]`, `[virtual]`

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited handle() method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters

in	*event*	the kind of event received

Return values

0	if the event was not used or understood
1	if the event was used and can be deleted

See Also

Fl_Event

Reimplemented from Fl_Group.
Reimplemented in Fl_Table_Row.

int Fl_Table::is_interactive_resize ( ) `[inline]`

Returns 1 if someone is interactively resizing a row or column.
You can currently call this only from within your callback().

int Fl_Table::is_selected (int *r*, int *c*)

See if the cell at row `r` and column `c` is selected.

Returns

1 if the cell is selected, 0 if not.

void Fl_Table::row_header (int *flag*) `[inline]`

Enables/disables showing the row headers.
1=enabled, 0=disabled. If changed, the table is redrawn.

void Fl_Table::row_height (int *row*, int *height*)

Sets the height of the specified row in pixels, and the table is redrawn.
callback() will be invoked with CONTEXT_RC_RESIZE if the row's height was actually changed, and when() is FL_WHEN_CHANGED.

void Fl_Table::row_height_all (int *height*) `[inline]`

Convenience method to set the height of all rows to the same value, in pixels.
The screen is redrawn.

void Fl_Table::row_resize (int *flag*) `[inline]`

Allows/disallows row resizing by the user.
1=allow interactive resizing, 0=disallow interactive resizing. Since interactive resizing is done via the row headers, row_header() must also be enabled to allow resizing.

void Fl_Table::row_resize_min (int *val*) `[inline]`

Sets the current row minimum resize value.
This is used to prevent the user from interactively resizing any row to be smaller than 'pixels'. Must be a value >=1.

int Fl_Table::scrollbar_size ( ) const `[inline]`

Gets the current size of the scrollbars' troughs, in pixels.

If this value is zero (default), this widget will use the Fl::scrollbar_size() value as the scrollbar's width.

Returns

Scrollbar size in pixels, or 0 if the global Fl::scrollbar_size() is being used.

See Also

Fl::scrollbar_size(int)

void Fl_Table::scrollbar_size (int *newSize*) `[inline]`

Sets the pixel size of the scrollbars' troughs to `newSize`, in pixels.

Normally you should not need this method, and should use Fl::scrollbar_size(int) instead to manage the size of ALL your widgets' scrollbars. This ensures your application has a consistent UI, is the default behavior, and is normally what you want.

Only use THIS method if you really need to override the global scrollbar size. The need for this should be rare.

Setting `newSize` to the special value of 0 causes the widget to track the global Fl::scrollbar_size(), which is the default.

Parameters

`in`	*newSize*	Sets the scrollbar size in pixels. If 0 (default), scrollbar size tracks the global Fl::scrollbar_size()

See Also

Fl::scrollbar_size()

void Fl_Table::set_selection (int *row_top*, int *col_left*, int *row_bot*, int *col_right*)

Sets the region of cells to be selected (highlighted).

So for instance, set_selection(0,0,0,0) selects the top/left cell in the table. And set_selection(0,0,1,1) selects the four cells in rows 0 and 1, column 0 and 1.

Parameters

`in`	*row_top*	Top row of selection area
`in`	*col_left*	Left column of selection area
`in`	*row_bot*	Bottom row of selection area
`in`	*col_right*	Right column of selection area

void Fl_Table::tab_cell_nav (int *val*) `[inline]`

Flag to control if Tab navigates table cells or not.

If on, Tab key navigates table cells. If off, Tab key navigates fltk widget focus. (default)

As of fltk 1.3, the default behavior of the Tab key is to navigate focus off of the current widget, and on to the next one. But in some applications, it's useful for Tab to be used to navigate cells in the Fl_Table.

Parameters

in	*val*	If `val` is 1, Tab key navigates cells in table, not fltk widgets. If `val` is 0, Tab key will advance focus to the next fltk widget (default), and does not navigate cells in table.

int Fl_Table::tab_cell_nav ( ) const [inline]

Get state of table's 'Tab' key cell navigation flag.

Returns

1 if Tab configured to navigate cells in table
0 to navigate widget focus (default)

See Also

tab_cell_nav(int)

void Fl_Table::table_box (Fl_Boxtype *val*) [inline]

Sets the kind of box drawn around the data table, the default being FL_NO_BOX.
Changing this value will cause the table to redraw.

void Fl_Table::top_row (int *row*) [inline]

Sets which row should be at the top of the table, scrolling as necessary, and the table is redrawn.
If the table cannot be scrolled that far, it is scrolled as far as possible.

int Fl_Table::top_row ( ) [inline]

Returns the current top row shown in the table.
This row may be partially obscured.

void Fl_Table::visible_cells (int & *r1*, int & *r2*, int & *c1*, int & *c2*) [inline]

Returns the range of row and column numbers for all visible and partially visible cells in the table.
These values can be used e.g. by your draw_cell() routine during CONTEXT_STARTPAGE to figure out what cells are about to be redrawn for the purposes of locking the data from a database before it's drawn.

```
       leftcol             rightcol
          :                   :
toprow .. .-------------------.
          |                   |
          |  V I S I B L E    |
          |                   |
          |    T A B L E      |
          |                   |
botrow .. '-------------------`
```

e.g. in a table where the visible rows are 5-20, and the visible columns are 100-120, then those variables would be:

- toprow = 5
- botrow = 20
- leftcol = 100
- rightcol = 120

void Fl_Table::when (Fl_When *flags*)

The Fl_Widget::when() function is used to set a group of flags, determining when the widget callback is called:

`FL_WHEN_CHANGED`	callback() will be called when rows or columns are resized (interactively or via col_width() or row_height()), passing CONTEXT_RC_RESIZE via callback_context().
`FL_WHEN_RELEASE`	callback() will be called during FL_RELEASE events, such as when someone releases a mouse button somewhere on the table.

The callback() routine is sent a TableContext that indicates the context the event occurred in, such as in a cell, in a header, or elsewhere on the table. When an event occurs in a cell or header, callback_row() and callback_col() can be used to determine the row and column. The callback can also look at the regular fltk event values (ie. Fl::event() and Fl::event_button()) to determine what kind of event is occurring.

The documentation for this class was generated from the following files:

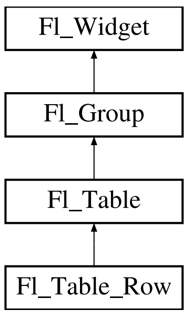
- Fl_Table.H
- Fl_Table.cxx

31.132 Fl_Table_Row Class Reference

A table with row selection capabilities.

`#include <Fl_Table_Row.H>`

Inheritance diagram for Fl_Table_Row:



Public Types

- enum **TableRowSelectMode** { **SELECT_NONE**, **SELECT_SINGLE**, **SELECT_MULTI** }

Public Member Functions

- void clear ()

  Clears the table to zero rows (rows(0)), zero columns (cols(0)), and clears any widgets (table->clear()) that were added with begin()/end() or add()/insert()/etc.
- Fl_Table_Row (int X, int Y, int W, int H, const char ∗l=0)

  The constructor for the Fl_Table_Row.
- int row_selected (int row)

  Checks to see if 'row' is selected.
- void rows (int val)

  Sets the number of rows in the table, and the table is redrawn.
- int **rows** ()
- void select_all_rows (int flag=1)

  This convenience function changes the selection state for all rows based on 'flag'.

- int select_row (int row, int flag=1)

  Changes the selection state for 'row', depending on the value of 'flag'.
- void type (TableRowSelectMode val)

  Sets the table selection mode.
- TableRowSelectMode **type** () const
- ∼Fl_Table_Row ()

  The destructor for the Fl_Table_Row.

Protected Member Functions

- int **find_cell** (TableContext context, int R, int C, int &X, int &Y, int &W, int &H)
- int handle (int event)

  Handles the specified event.

Additional Inherited Members

31.132.1 Detailed Description

A table with row selection capabilities.

This class implements a simple table with the ability to select rows. This widget is similar to an Fl_Browser with columns. Most methods of importance will be found in the Fl_Table widget, such as Fl_Table::rows() and Fl_Table::cols().

To be useful it must be subclassed and at minimum the draw_cell() method must be overridden to provide the content of the cells. This widget does *not* manage the cell's data content; it is up to the parent class's draw_cell() method override to provide this.

Events on the cells and/or headings generate callbacks when they are clicked by the user. You control when events are generated based on the values you supply for Fl_Table::when().

31.132.2 Constructor & Destructor Documentation

Fl_Table_Row::Fl_Table_Row (int *X*, int *Y*, int *W*, int *H*, const char ∗ *l* = *0*) `[inline]`

The constructor for the Fl_Table_Row.

This creates an empty table with no rows or columns, with headers and row/column resize behavior disabled.

Fl_Table_Row::∼Fl_Table_Row ( ) `[inline]`

The destructor for the Fl_Table_Row.

Destroys the table and its associated widgets.

31.132.3 Member Function Documentation

void Fl_Table_Row::clear ( ) `[inline]`, `[virtual]`

Clears the table to zero rows (rows(0)), zero columns (cols(0)), and clears any widgets (table->clear()) that were added with begin()/end() or add()/insert()/etc.

See Also

rows(int), cols(int)

Reimplemented from Fl_Table.

int Fl_Table_Row::handle (int *event*) `[protected]`, `[virtual]`

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited handle() method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters

`in`	*event*	the kind of event received

Return values

0	if the event was not used or understood
1	if the event was used and can be deleted

See Also

Fl_Event

Reimplemented from Fl_Table.

int Fl_Table_Row::row_selected (int *row*)

Checks to see if 'row' is selected.

Returns 1 if selected, 0 if not. You can change the selection of a row by clicking on it, or by using select_row(row, flag)

void Fl_Table_Row::select_all_rows (int *flag* = `1`)

This convenience function changes the selection state for *all* rows based on 'flag'.

0=deselect, 1=select, 2=toggle existing state.

int Fl_Table_Row::select_row (int *row*, int *flag* = `1`)

Changes the selection state for 'row', depending on the value of 'flag'.

0=deselected, 1=select, 2=toggle existing state.

void Fl_Table_Row::type (TableRowSelectMode *val*)

Sets the table selection mode.

- `Fl_Table_Row::SELECT_NONE` - No selection allowed
- `Fl_Table_Row::SELECT_SINGLE` - Only single rows can be selected
- `Fl_Table_Row::SELECT_MULTI` - Multiple rows can be selected

The documentation for this class was generated from the following files:

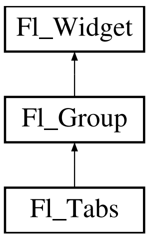
- Fl_Table_Row.H
- Fl_Table_Row.cxx

31.133 Fl_Tabs Class Reference

The Fl_Tabs widget is the "file card tabs" interface that allows you to put lots and lots of buttons and switches in a panel, as popularized by many toolkits.

`#include <Fl_Tabs.H>`

Inheritance diagram for Fl_Tabs:



Public Member Functions

- void client_area (int &rx, int &ry, int &rw, int &rh, int tabh=0)

  Returns the position and size available to be used by its children.
- Fl_Tabs (int, int, int, int, const char ∗=0)

  Creates a new Fl_Tabs widget using the given position, size, and label string.
- int handle (int)

  Handles the specified event.
- Fl_Widget ∗ push () const

  Returns the tab group for the tab the user has currently down-clicked on and remains over until FL_RELEASE.
- int push (Fl_Widget ∗)

  This is called by the tab widget's handle() method to set the tab group widget the user last FL_PUSH'ed on.
- Fl_Widget ∗ value ()

  Gets the currently visible widget/tab.
- int value (Fl_Widget ∗)

  Sets the widget to become the current visible widget/tab.
- Fl_Widget ∗ which (int event_x, int event_y)

  Return the widget of the tab the user clicked on at `event_x` */* `event_y`*.*

Protected Member Functions

- void draw ()

  Draws the widget.
- void **redraw_tabs** ()

Additional Inherited Members

31.133.1 Detailed Description

The Fl_Tabs widget is the "file card tabs" interface that allows you to put lots and lots of buttons and switches in a panel, as popularized by many toolkits.

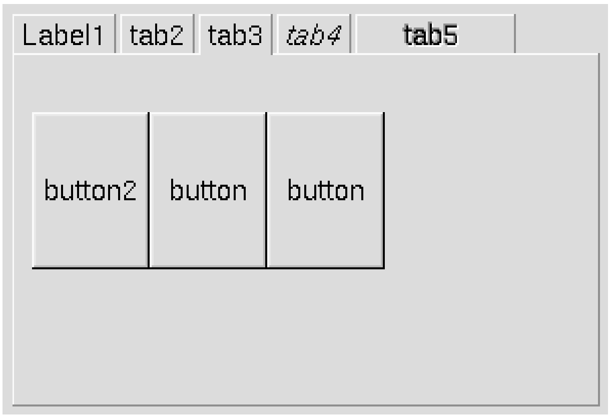


Figure 31.37: Fl_Tabs

Clicking the tab makes a child visible() by calling show() on it, and all other children are made invisible by calling hide() on them. Usually the children are Fl_Group widgets containing several widgets themselves.

Each child makes a card, and its label() is printed on the card tab, including the label font and style. The selection color of that child is used to color the tab, while the color of the child determines the background color of the pane.

The size of the tabs is controlled by the bounding box of the children (there should be some space between the children and the edge of the Fl_Tabs), and the tabs may be placed ”inverted” on the bottom - this is determined by which gap is larger. It is easiest to lay this out in fluid, using the fluid browser to select each child group and resize them until the tabs look the way you want them to.

The background area behind and to the right of the tabs is ”transparent”, exposing the background detail of the parent. The value of Fl_Tabs::box() does not affect this area. So if Fl_Tabs is resized by itself without the parent, force the appropriate parent (visible behind the tabs) to redraw() to prevent artifacts.

See ”Resizing Caveats” below on how to keep tab heights constant. See ”Callback's Use Of when()” on how to control the details of how clicks invoke the callback().

A typical use of the Fl_Tabs widget:

```
// Typical use of Fl_Tabs
Fl_Tabs *tabs = new Fl_Tabs(10,10,300,200);
{
    Fl_Group *grp1 = new Fl_Group(20,30,280,170,"Tab1");
    {
        ..widgets that go in tab#1..
    }
    grp1->end();
    Fl_Group *grp2 = new Fl_Group(20,30,280,170,"Tab2");
    {
        ..widgets that go in tab#2..
    }
    grp2->end();
}
tabs->end();
```

Default Appearance

The appearance of each ”tab” is taken from the label() and color() of the child group corresponding to that ”tab” and panel. Where the ”tabs” appear depends on the position and size of the child groups that make up the panels within the Fl_Tab, i.e. whether there is more space above or below them. The height of the ”tabs” depends on how much free space is available.

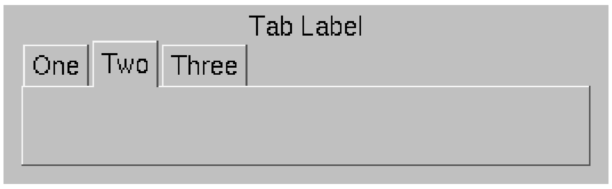


Figure 31.38: Fl_Tabs Default Appearance

Highlighting The Selected Tab

The selected "tab" can be highlighted further by setting the selection_color() of the Fl_Tab itself, e.g.

```
..
tabs = new Fl_Tabs(..);
tabs->selection_color(FL_DARK3);
..
```

The result of the above looks like:

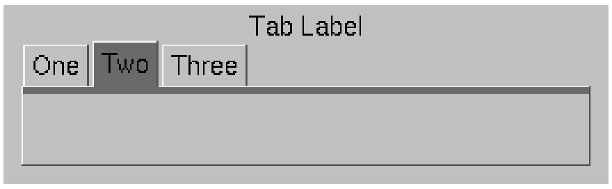


Figure 31.39: Highlighting the selected tab

Uniform Tab and Panel Appearance

In order to have uniform tab and panel appearance, not only must the color() and selection_color() for each child group be set, but also the selection_color() of the Fl_Tab itself any time a new "tab" is selected. This can be achieved within the Fl_Tab callback, e.g.

```
void MyTabCallback(Fl_Widget *w, void*) {
  Fl_Tabs *tabs = (Fl_Tabs*)w;
  // When tab changed, make sure it has same color as its group
  tabs->selection_color( (tab->value())->color() );
}
..
int main(..) {
  // Define tabs widget
  tabs = new Fl_Tabs(..);
  tabs->callback(MyTabCallback);

  // Create three tabs each colored differently
  grp1 = new Fl_Group(.. "One");
   grp1->color(9);
   grp1->selection_color(9);
  grp1->end();

  grp2 = new Fl_Group(.. "Two");
   grp2->color(10);
   grp2->selection_color(10);
  grp2->end();

  grp3 = new Fl_Group(.. "Three");
   grp3->color(14);
   grp3->selection_color(14);
  grp3->end();
  ..
  // Make sure default tab has same color as its group
  tabs->selection_color( (tab->value())->color() );
```

```
    ..
    return Fl::run();
}
```

The result of the above looks like:

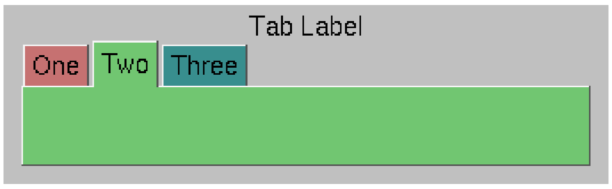


Figure 31.40: Fl_Tabs with uniform colors

Resizing Caveats

When Fl_Tabs is resized vertically, the default behavior scales the tab's height as well as its children. To keep the tab height constant during resizing, set the tab widget's resizable() to one of the tab's child groups, i.e.

```
tabs = new Fl_Tabs(..);
grp1 = new Fl_Group(..);
..
grp2 = new Fl_Group(..);
..
tabs->end();
tabs->resizable(grp1);        // keeps tab height constant
```

Callback's Use Of when()

As of FLTK 1.3.3, Fl_Tabs() supports the following flags for when():

- FL_WHEN_NEVER – callback never invoked (all flags off)
- FL_WHEN_CHANGED – if flag set, invokes callback when a tab has been changed (on click or keyboard navigation)
- FL_WHEN_NOT_CHANGED – if flag set, invokes callback when the tabs remain unchanged (on click or keyboard navigation)
- FL_WHEN_RELEASE – if flag set, invokes callback on RELEASE of mouse button or keyboard navigation

Notes:

1. The above flags can be logically OR-ed (|) or added (+) to combine behaviors.
2. The default value for when() is FL_WHEN_RELEASE (inherited from Fl_Widget).
3. If FL_WHEN_RELEASE is the *only* flag specified, the behavior will be as if (FL_WHEN_RELEASE|FL_WHEN_CHANGED) was specified.
4. The value of changed() will be valid during the callback.
5. If both FL_WHEN_CHANGED and FL_WHEN_NOT_CHANGED are specified, the callback is invoked whether the tab has been changed or not. The changed() method can be used to determine the cause.
6. FL_WHEN_NOT_CHANGED can happen if someone clicks on an already selected tab, or if a keyboard navigation attempt results in no change to the tabs, such as using the arrow keys while at the left or right end of the tabs.

31.133.2 Constructor & Destructor Documentation

Fl_Tabs::Fl_Tabs (int *X*, int *Y*, int *W*, int *H*, const char * *l* = 0)

Creates a new Fl_Tabs widget using the given position, size, and label string.

The default boxtype is FL_THIN_UP_BOX.

Use add(Fl_Widget*) to add each child, which are usually Fl_Group widgets. The children should be sized to stay away from the top or bottom edge of the Fl_Tabs widget, which is where the tabs will be drawn.

All children of Fl_Tabs should have the same size and exactly fit on top of each other. They should only leave space above or below where the tabs will go, but not on the sides. If the first child of Fl_Tabs is set to ”resizable()”, the riders will not resize when the tabs are resized.

The destructor *also deletes all the children*. This allows a whole tree to be deleted at once, without having to keep a pointer to all the children in the user code. A kludge has been done so the Fl_Tabs and all of its children can be automatic (local) variables, but you must declare the Fl_Tabs widget *first* so that it is destroyed last.

31.133.3 Member Function Documentation

void Fl_Tabs::client_area (int & *rx*, int & *ry*, int & *rw*, int & *rh*, int *tabh* = 0)

Returns the position and size available to be used by its children.

If there isn't any child yet the `tabh` parameter will be used to calculate the return values. This assumes that the children's labelsize is the same as the Fl_Tabs' labelsize and adds a small border.

If there are already children, the values of child(0) are returned, and `tabh` is ignored.

Note

Children should always use the same positions and sizes.

`tabh` can be one of

- 0: calculate label size, tabs on top
- -1: calculate label size, tabs on bottom
- $>$ 0: use given `tabh` value, tabs on top (height = tabh)
- $<$ -1: use given `tabh` value, tabs on bottom (height = -tabh)

Parameters

in	*tabh*	position and optional height of tabs (see above)
out	*rx,ry,rw,rh*	(x,y,w,h) of client area for children

Since

FLTK 1.3.0

void Fl_Tabs::draw ( ) `[protected], [virtual]`

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call redraw() instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw() method*, e.g. for an embedded scrollbar, you can do it (because draw() is virtual) like this:

```
Fl_Widget *s = &scroll;              // scroll is an embedded Fl_Scrollbar
s->draw();                 // calls Fl_Scrollbar::draw()
```

Reimplemented from Fl_Group.

int Fl_Tabs::handle (int *event*) `[virtual]`

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited handle() method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters

`in`	*event*	the kind of event received

Return values

0	if the event was not used or understood
1	if the event was used and can be deleted

See Also

Fl_Event

Reimplemented from Fl_Group.

Fl_Widget∗ Fl_Tabs::push ( ) const `[inline]`

Returns the tab group for the tab the user has currently down-clicked on and remains over until FL_RELEASE.

Otherwise, returns NULL.

While the user is down-clicked on a tab, the return value is the tab group for that tab. But as soon as the user releases, or drags off the tab with the button still down, the return value will be NULL.

See Also

push(Fl_Widget∗).

int Fl_Tabs::push (Fl_Widget ∗ *o*)

This is called by the tab widget's handle() method to set the tab group widget the user last FL_PUSH'ed on.

Set back to zero on FL_RELEASE.

As of this writing, the value is mainly used by draw_tab() to determine whether or not to draw a 'down' box for the tab when it's clicked, and to turn it off if the user drags off it.

See Also

push().

Fl_Widget ∗ Fl_Tabs::value ( )

Gets the currently visible widget/tab.

The value() is the first visible child (or the last child if none are visible) and this also hides any other children. This allows the tabs to be deleted, moved to other groups, and show()/hide() called without it screwing up.

int Fl_Tabs::value (Fl_Widget ∗ *newvalue*)

Sets the widget to become the current visible widget/tab.
Setting the value hides all other children, and makes this one visible, if it is really a child.

Returns

1 if there was a change (new value different from previous),
0 if there was no change (new value already set)

Fl_Widget ∗ Fl_Tabs::which (int *event_x,* int *event_y*)

Return the widget of the tab the user clicked on at `event_x` / `event_y`.
This is used for event handling (clicks) and by fluid to pick tabs.

Returns

The child widget of the tab the user clicked on, or
0 if there are no children or if the event is outside of the tabs area.

The documentation for this class was generated from the following files:

- Fl_Tabs.H
- Fl_Tabs.cxx

31.134 Fl_Text_Buffer Class Reference

This class manages Unicode text displayed in one or more Fl_Text_Display widgets.
`#include <Fl_Text_Buffer.H>`

Public Member Functions

- void add_modify_callback (Fl_Text_Modify_Cb bufModifiedCB, void ∗cbArg)
  Adds a callback function that is called whenever the text buffer is modified.
- void add_predelete_callback (Fl_Text_Predelete_Cb bufPredelCB, void ∗cbArg)
  Adds a callback routine to be called before text is deleted from the buffer.
- const char ∗ address (int pos) const
  Convert a byte offset in buffer into a memory address.
- char ∗ address (int pos)
  Convert a byte offset in buffer into a memory address.
- void append (const char ∗t)
  Appends the text string to the end of the buffer.
- int appendfile (const char ∗file, int buflen=128 ∗1024)
  Appends the named file to the end of the buffer.
- char byte_at (int pos) const
  Returns the raw byte at the specified position pos in the buffer.
- void call_modify_callbacks ()
  Calls all modify callbacks that have been registered using the add_modify_callback() method.
- void call_predelete_callbacks ()
  Calls the stored pre-delete callback procedure(s) for this buffer to update the changed area(s) on the screen and any other listeners.
- void canUndo (char flag=1)

Lets the undo system know if we can undo changes.

- unsigned int char_at (int pos) const

  Returns the character at the specified position `pos` *in the buffer.*
- void copy (Fl_Text_Buffer ∗fromBuf, int fromStart, int fromEnd, int toPos)

  Copies text from another Fl_Text_Buffer to this one.
- int count_displayed_characters (int lineStartPos, int targetPos) const

  Count the number of displayed characters between buffer position `lineStartPos` *and* `targetPos`*.*
- int count_lines (int startPos, int endPos) const

  Counts the number of newlines between `startPos` *and* `endPos` *in buffer.*
- int findchar_backward (int startPos, unsigned int searchChar, int ∗foundPos) const

  Search backwards in buffer `buf` *for character* `searchChar`*, starting with the character before* `start-Pos`*, returning the result in* `foundPos`*.*
- int findchar_forward (int startPos, unsigned searchChar, int ∗foundPos) const

  Finds the next occurrence of the specified character.
- Fl_Text_Buffer (int requestedSize=0, int preferredGapSize=1024)

  Create an empty text buffer of a pre-determined size.
- void highlight (int start, int end)

  Highlights the specified text within the buffer.
- int highlight ()

  Returns the highlighted text.
- int highlight_position (int ∗start, int ∗end)

  Highlights the specified text between `start` *and* `end` *within the buffer.*
- const Fl_Text_Selection ∗ highlight_selection () const

  Returns the current highlight selection.
- char ∗ highlight_text ()

  Returns the highlighted text.
- void insert (int pos, const char ∗text)

  Inserts null-terminated string `text` *at position* `pos`*.*
- int insertfile (const char ∗file, int pos, int buflen=128 ∗1024)

  Inserts a file at the specified position.
- int length () const

  Returns the number of bytes in the buffer.
- int line_end (int pos) const

  Finds and returns the position of the end of the line containing position `pos` *(which is either a pointer to the newline character ending the line or a pointer to one character beyond the end of the buffer).*
- int line_start (int pos) const

  Returns the position of the start of the line containing position `pos`*.*
- char ∗ line_text (int pos) const

  Returns the text from the entire line containing the specified character position.
- int loadfile (const char ∗file, int buflen=128 ∗1024)

  Loads a text file into the buffer.
- int next_char (int ix) const

  Returns the index of the next character.
- int **next_char_clipped** (int ix) const
- int outputfile (const char ∗file, int start, int end, int buflen=128 ∗1024)

  Writes the specified portions of the text buffer to a file.
- int prev_char (int ix) const

Returns the index of the previous character.

- int **prev_char_clipped** (int ix) const
- const Fl_Text_Selection ∗ primary_selection () const

  Returns the primary selection.
- Fl_Text_Selection ∗ primary_selection ()

  Returns the primary selection.
- void remove (int start, int end)

  Deletes a range of characters in the buffer.
- void remove_modify_callback (Fl_Text_Modify_Cb bufModifiedCB, void ∗cbArg)

  Removes a modify callback.
- void remove_predelete_callback (Fl_Text_Predelete_Cb predelCB, void ∗cbArg)

  Removes a callback routine `bufPreDeleteCB` associated with argument `cbArg` to be called before text is deleted from the buffer.
- void remove_secondary_selection ()

  Removes the text from the buffer corresponding to the secondary text selection object.
- void remove_selection ()

  Removes the text in the primary selection.
- void replace (int start, int end, const char ∗text)

  Deletes the characters between `start` and `end`, and inserts the null-terminated string `text` in their place in the buffer.
- void replace_secondary_selection (const char ∗text)

  Replaces the text from the buffer corresponding to the secondary text selection object with the new string `text`.
- void replace_selection (const char ∗text)

  Replaces the text in the primary selection.
- int rewind_lines (int startPos, int nLines)

  Finds and returns the position of the first character of the line `nLines` backwards from `startPos` (not counting the character pointed to by `startpos` if that is a newline) in the buffer.
- int savefile (const char ∗file, int buflen=128 ∗1024)

  Saves a text file from the current buffer.
- int search_backward (int startPos, const char ∗searchString, int ∗foundPos, int matchCase=0) const

  Search backwards in buffer for string `searchString`, starting with the character at `startPos`, returning the result in `foundPos`.
- int search_forward (int startPos, const char ∗searchString, int ∗foundPos, int matchCase=0) const

  Search forwards in buffer for string `searchString`, starting with the character `startPos`, and returning the result in `foundPos`.
- void secondary_select (int start, int end)

  Selects a range of characters in the secondary selection.
- int secondary_selected ()

  Returns a non-zero value if text has been selected in the secondary text selection, 0 otherwise.
- const Fl_Text_Selection ∗ secondary_selection () const

  Returns the secondary selection.
- int secondary_selection_position (int ∗start, int ∗end)

  Returns the current selection in the secondary text selection object.
- char ∗ secondary_selection_text ()

  Returns the text in the secondary selection.
- void secondary_unselect ()

  Clears any selection in the secondary text selection object.

- void select (int start, int end)

  Selects a range of characters in the buffer.
- int selected () const

  Returns a non-zero value if text has been selected, 0 otherwise.
- int selection_position (int ∗start, int ∗end)

  Gets the selection position.
- char ∗ selection_text ()

  Returns the currently selected text.
- int skip_displayed_characters (int lineStartPos, int nChars)

  Count forward from buffer position `startPos` in displayed characters.
- int skip_lines (int startPos, int nLines)

  Finds the first character of the line `nLines` forward from `startPos` in the buffer and returns its position.
- int tab_distance () const

  Gets the tab width.
- void tab_distance (int tabDist)

  Set the hardware tab distance (width) used by all displays for this buffer, and used in computing offsets for rectangular selection operations.
- char ∗ text () const

  Get a copy of the entire contents of the text buffer.
- void text (const char ∗text)

  Replaces the entire contents of the text buffer.
- char ∗ text_range (int start, int end) const

  Get a copy of a part of the text buffer.
- int undo (int ∗cp=0)

  Undo text modification according to the undo variables or insert text from the undo buffer.
- void unhighlight ()

  Unhighlights text in the buffer.
- void unselect ()

  Cancels any previous selection on the primary text selection object.
- int utf8_align (int) const

  Align an index into the buffer to the current or previous UTF-8 boundary.
- int word_end (int pos) const

  Returns the position corresponding to the end of the word.
- int word_start (int pos) const

  Returns the position corresponding to the start of the word.
- ∼Fl_Text_Buffer ()

  Frees a text buffer.

Public Attributes

- int input_file_was_transcoded

  true if the loaded file has been transcoded to UTF-8.
- void(∗ transcoding_warning_action)(Fl_Text_Buffer ∗)

  Pointer to a function called after reading a non UTF-8 encoded file.

Static Public Attributes

- static const char ∗ file_encoding_warning_message

    This message may be displayed using the fl_alert() function when a file which was not UTF-8 encoded is input.

Protected Member Functions

- void call_modify_callbacks (int pos, int nDeleted, int nInserted, int nRestyled, const char ∗deleted-Text) const

    Calls the stored modify callback procedure(s) for this buffer to update the changed area(s) on the screen and any other listeners.
- void call_predelete_callbacks (int pos, int nDeleted) const

    Calls the stored pre-delete callback procedure(s) for this buffer to update the changed area(s) on the screen and any other listeners.
- int insert_ (int pos, const char ∗text)

    Internal (non-redisplaying) version of insert().
- void move_gap (int pos)

    Move the gap to start at a new position.
- void reallocate_with_gap (int newGapStart, int newGapLen)

    Reallocates the text storage in the buffer to have a gap starting at `newGapStart` and a gap size of `new-GapLen`, preserving the buffer's current contents.
- void redisplay_selection (Fl_Text_Selection ∗oldSelection, Fl_Text_Selection ∗newSelection) const

    Calls the stored redisplay procedure(s) for this buffer to update the screen for a change in a selection.
- void remove_ (int start, int end)

    Internal (non-redisplaying) version of remove().
- void remove_selection_ (Fl_Text_Selection ∗sel)

    Removes the text from the buffer corresponding to `sel`.
- void replace_selection_ (Fl_Text_Selection ∗sel, const char ∗text)

    Replaces the `text` in selection `sel`.
- char ∗ **selection_text_** (Fl_Text_Selection ∗sel) const
- void update_selections (int pos, int nDeleted, int nInserted)

    Updates all of the selections in the buffer for changes in the buffer's text.

Protected Attributes

- char ∗ mBuf

    allocated memory where the text is stored
- char mCanUndo

    if this buffer is used for attributes, it must not do any undo calls
- void ∗∗ mCbArgs

    caller arguments for modifyProcs above
- int mCursorPosHint

    hint for reasonable cursor position after a buffer modification operation
- int mGapEnd

    points to the first character after the gap
- int mGapStart

    points to the first character of the gap
- Fl_Text_Selection mHighlight

highlighted areas
- int mLength

  length of the text in the buffer (the length of the buffer itself must be calculated: gapEnd - gapStart + length)
- Fl_Text_Modify_Cb ∗ mModifyProcs

  procedures to call when buffer is modified to redisplay contents
- int mNModifyProcs

  number of modify-redisplay procs attached
- int mNPredeleteProcs

  number of pre-delete procs attached
- void ∗∗ mPredeleteCbArgs

  caller argument for pre-delete proc above
- Fl_Text_Predelete_Cb ∗ mPredeleteProcs

  procedure to call before text is deleted from the buffer; at most one is supported.
- int mPreferredGapSize

  the default allocation for the text gap is 1024 bytes and should only be increased if frequent and large changes in buffer size are expected
- Fl_Text_Selection mPrimary

  highlighted areas
- Fl_Text_Selection mSecondary

  highlighted areas
- int mTabDist

  equiv.

31.134.1 Detailed Description

This class manages Unicode text displayed in one or more Fl_Text_Display widgets.

All text in Fl_Text_Buffer must be encoded in UTF-8. All indices used in the function calls must be aligned to the start of a UTF-8 sequence. All indices and pointers returned will be aligned. All functions that return a single character will return that in an unsiged int in UCS-4 encoding.

The Fl_Text_Buffer class is used by the Fl_Text_Display and Fl_Text_Editor to manage complex text data and is based upon the excellent NEdit text editor engine - see `http://www.nedit.org/`.

31.134.2 Constructor & Destructor Documentation

Fl_Text_Buffer::Fl_Text_Buffer (int *requestedSize* = 0, int *preferredGapSize* = 1024)

Create an empty text buffer of a pre-determined size.

Parameters

requestedSize	use this to avoid unnecessary re-allocation if you know exactly how much the buffer will need to hold
preferredGap-Size	Initial size for the buffer gap (empty space in the buffer where text might be inserted if the user is typing sequential characters)

31.134.3 Member Function Documentation

void Fl_Text_Buffer::add_modify_callback (Fl_Text_Modify_Cb *bufModifiedCB*, void ∗ *cbArg*)

Adds a callback function that is called whenever the text buffer is modified.

The callback function is declared as follows:

```
typedef void (*Fl_Text_Modify_Cb)(int pos, int nInserted, int nDeleted,
   int nRestyled, const char* deletedText,
   void* cbArg);
```

const char∗ Fl_Text_Buffer::address (int *pos*) const `[inline]`

Convert a byte offset in buffer into a memory address.

Parameters

pos	byte offset into buffer

Returns

byte offset converted to a memory address

char∗ Fl_Text_Buffer::address (int *pos*) `[inline]`

Convert a byte offset in buffer into a memory address.
Parameters

pos	byte offset into buffer

Returns

byte offset converted to a memory address

void Fl_Text_Buffer::append (const char ∗ *t*) `[inline]`

Appends the text string to the end of the buffer.
Parameters

t	UTF-8 encoded and nul terminated text

int Fl_Text_Buffer::appendfile (const char ∗ *file*, int *buflen* = `128*1024`) `[inline]`

Appends the named file to the end of the buffer.
See also insertfile().

char Fl_Text_Buffer::byte_at (int *pos*) const

Returns the raw byte at the specified position pos in the buffer.
Positions start at 0.
Parameters

pos	byte offset into buffer

Returns

unencoded raw byte

unsigned int Fl_Text_Buffer::char_at (int *pos*) const

Returns the character at the specified position `pos` in the buffer.
Positions start at 0.
Parameters

pos	byte offset into buffer, `pos` must be at a UTF-8 character boundary

Returns

Unicode UCS-4 encoded character

void Fl_Text_Buffer::copy (Fl_Text_Buffer ∗ *fromBuf*, int *fromStart*, int *fromEnd*, int *toPos*)

Copies text from another Fl_Text_Buffer to this one.

Parameters

fromBuf	source text buffer, may be the same as this
fromStart	byte offset into buffer
fromEnd	byte offset into buffer
toPos	destination byte offset into buffer

int Fl_Text_Buffer::count_displayed_characters (int *lineStartPos,* int *targetPos*) const

Count the number of displayed characters between buffer position `lineStartPos` and `targetPos`.

Displayed characters are the characters shown on the screen to represent characters in the buffer, where tabs and control characters are expanded.

int Fl_Text_Buffer::count_lines (int *startPos,* int *endPos*) const

Counts the number of newlines between `startPos` and `endPos` in buffer.

The character at position `endPos` is not counted.

int Fl_Text_Buffer::findchar_backward (int *startPos,* unsigned int *searchChar,* int ∗ *foundPos*) const

Search backwards in buffer `buf` for character `searchChar`, starting with the character *before* `startPos`, returning the result in `foundPos`.

Returns 1 if found, 0 if not. The difference between this and search_backward() is that it's optimized for single characters. The overall performance of the text widget is dependent on its ability to count lines quickly, hence searching for a single character: newline.

Parameters

startPos	byte offset to start position
searchChar	UCS-4 character that we want to find
foundPos	byte offset where the character was found

Returns

1 if found, 0 if not

int Fl_Text_Buffer::findchar_forward (int *startPos,* unsigned *searchChar,* int ∗ *foundPos*) const

Finds the next occurrence of the specified character.

Search forwards in buffer for character `searchChar`, starting with the character `startPos`, and returning the result in `foundPos`. Returns 1 if found, 0 if not. The difference between this and search_forward() is that it's optimized for single characters. The overall performance of the text widget is dependent on its ability to count lines quickly, hence searching for a single character: newline.

Parameters

startPos	byte offset to start position
searchChar	UCS-4 character that we want to find
foundPos	byte offset where the character was found

Returns

1 if found, 0 if not

int Fl_Text_Buffer::highlight ( ) [inline]

Returns the highlighted text.

When you are done with the text, free it using the free() function.

char * Fl_Text_Buffer::highlight_text ( )

Returns the highlighted text.

When you are done with the text, free it using the free() function.

void Fl_Text_Buffer::insert (int *pos*, const char * *text*)

Inserts null-terminated string `text` at position `pos`.

Parameters

pos	insertion position as byte offset (must be UTF-8 character aligned)
text	UTF-8 encoded and nul terminated text

int Fl_Text_Buffer::insert_ (int *pos*, const char * *text*) [protected]

Internal (non-redisplaying) version of insert().

Returns the length of text inserted (this is just strlen(`text`), however this calculation can be expensive and the length will be required by any caller who will continue on to call redisplay). `pos` must be contiguous with the existing text in the buffer (i.e. not past the end).

Returns

the number of bytes inserted

int Fl_Text_Buffer::insertfile (const char * *file*, int *pos*, int *buflen* = 128*1024)

Inserts a file at the specified position.

Returns

- 0 on success
- non-zero on error (strerror() contains reason)
- 1 indicates open for read failed (no data loaded)
- 2 indicates error occurred while reading data (data was partially loaded)

File can be UTF-8 or CP1252 encoded. If the input file is not UTF-8 encoded, the Fl_Text_Buffer widget will contain data transcoded to UTF-8. By default, the message Fl_Text_Buffer::file_encoding_warning_message will warn the user about this.

See Also

input_file_was_transcoded and transcoding_warning_action.

int Fl_Text_Buffer::length ( ) const [inline]

Returns the number of bytes in the buffer.

Returns

size of text in bytes

int Fl_Text_Buffer::line_end (int *pos*) const

Finds and returns the position of the end of the line containing position `pos` (which is either a pointer to the newline character ending the line or a pointer to one character beyond the end of the buffer).

Parameters

pos	byte index into buffer

Returns

byte offset to line end

int Fl_Text_Buffer::line_start (int *pos*) const

Returns the position of the start of the line containing position `pos`.
Parameters

pos	byte index into buffer

Returns

byte offset to line start

char ∗ Fl_Text_Buffer::line_text (int *pos*) const

Returns the text from the entire line containing the specified character position.
When you are done with the text, free it using the free() function.
Parameters

pos	byte index into buffer

Returns

copy of UTF-8 text, must be free'd

int Fl_Text_Buffer::loadfile (const char ∗ *file*, int *buflen* = `128∗1024`) `[inline]`

Loads a text file into the buffer.
See also insertfile().

int Fl_Text_Buffer::next_char (int *ix*) const

Returns the index of the next character.
Parameters

ix	index to the current character

int Fl_Text_Buffer::outputfile (const char ∗ *file*, int *start*, int *end*, int *buflen* = `128∗1024`)

Writes the specified portions of the text buffer to a file.
Returns

- 0 on success
- non-zero on error (strerror() contains reason)
- 1 indicates open for write failed (no data saved)
- 2 indicates error occurred while writing data (data was partially saved)

See Also

savefile(const char ∗file, int buflen)

int Fl_Text_Buffer::prev_char (int *ix*) const

Returns the index of the previous character.

Parameters

ix	index to the current character

void Fl_Text_Buffer::remove (int *start*, int *end*)

Deletes a range of characters in the buffer.
Parameters

start	byte offset to first character to be removed
end	byte offset to character after last character to be removed

void Fl_Text_Buffer::remove_ (int *start*, int *end*) `[protected]`

Internal (non-redisplaying) version of remove().

Removes the contents of the buffer between `start` and `end` (and moves the gap to the site of the delete).

void Fl_Text_Buffer::replace (int *start*, int *end*, const char ∗ *text*)

Deletes the characters between `start` and `end`, and inserts the null-terminated string `text` in their place in the buffer.
Parameters

start	byte offset to first character to be removed and new insert position
end	byte offset to character after last character to be removed
text	UTF-8 encoded and nul terminated text

int Fl_Text_Buffer::rewind_lines (int *startPos*, int *nLines*)

Finds and returns the position of the first character of the line `nLines` backwards from `startPos` (not counting the character pointed to by `startpos` if that is a newline) in the buffer.

`nLines` == 0 means find the beginning of the line.

int Fl_Text_Buffer::savefile (const char ∗ *file*, int *buflen* = `128*1024`) `[inline]`

Saves a text file from the current buffer.

Returns

- 0 on success
- non-zero on error (strerror() contains reason)
- 1 indicates open for write failed (no data saved)
- 2 indicates error occurred while writing data (data was partially saved)

See Also

outputfile(const char ∗file, int start, int end, int buflen)

int Fl_Text_Buffer::search_backward (int *startPos*, const char ∗ *searchString*, int ∗ *foundPos*, int *matchCase* = `0`) const

Search backwards in buffer for string `searchString`, starting with the character *at* `startPos`, returning the result in `foundPos`.

Returns 1 if found, 0 if not.

Parameters

startPos	byte offset to start position
searchString	UTF-8 string that we want to find
foundPos	byte offset where the string was found
matchCase	if set, match character case

Returns

1 if found, 0 if not

int Fl_Text_Buffer::search_forward (int *startPos,* const char ∗ *searchString,* int ∗ *foundPos,* int *matchCase = `0`*) const

Search forwards in buffer for string `searchString`, starting with the character `startPos`, and returning the result in `foundPos`.

Returns 1 if found, 0 if not.

Parameters

startPos	byte offset to start position
searchString	UTF-8 string that we want to find
foundPos	byte offset where the string was found
matchCase	if set, match character case

Returns

1 if found, 0 if not

char ∗ Fl_Text_Buffer::secondary_selection_text ( )

Returns the text in the secondary selection.

When you are done with the text, free it using the free() function.

char ∗ Fl_Text_Buffer::selection_text ( )

Returns the currently selected text.

When you are done with the text, free it using the free() function.

int Fl_Text_Buffer::skip_displayed_characters (int *lineStartPos,* int *nChars*)

Count forward from buffer position `startPos` in displayed characters.

Displayed characters are the characters shown on the screen to represent characters in the buffer, where tabs and control characters are expanded.

Parameters

lineStartPos	byte offset into buffer
nChars	number of bytes that are sent to the display

Returns

byte offset in input after all output bytes are sent

int Fl_Text_Buffer::tab_distance ( ) const `[inline]`

Gets the tab width.

The tab width is measured in characters. The pixel position is calculated using an average character width.

char ∗ Fl_Text_Buffer::text ( ) const

Get a copy of the entire contents of the text buffer.

Memory is allocated to contain the returned string, which the caller must free.

Returns

newly allocated text buffer - must be free'd, text is UTF-8

void Fl_Text_Buffer::text (const char ∗ *text*)

Replaces the entire contents of the text buffer.

Parameters

text	Text must be valid UTF-8. If null, an empty string is substituted.

char ∗ Fl_Text_Buffer::text_range (int *start,* int *end*) const

Get a copy of a part of the text buffer.

Return a copy of the text between `start` and `end` character positions from text buffer `buf`. Positions start at 0, and the range does not include the character pointed to by `end`. When you are done with the text, free it using the free() function.

Parameters

start	byte offset to first character
end	byte offset after last character in range

Returns

newly allocated text buffer - must be free'd, text is UTF-8

int Fl_Text_Buffer::word_end (int *pos*) const

Returns the position corresponding to the end of the word.

Parameters

pos	byte index into buffer

Returns

byte offset to word end

int Fl_Text_Buffer::word_start (int *pos*) const

Returns the position corresponding to the start of the word.

Parameters

pos	byte index into buffer

Returns

byte offset to word start

31.134.4 Member Data Documentation

const char ∗ Fl_Text_Buffer::file_encoding_warning_message [static]

Initial value:

```
=
"Displayed text contains the UTF-8 transcoding\n"
"of the input file which was not UTF-8 encoded.\n"
"Some changes may have occurred."
```

This message may be displayed using the fl_alert() function when a file which was not UTF-8 encoded is input.

Fl_Text_Predelete_Cb∗ Fl_Text_Buffer::mPredeleteProcs [protected]

procedure to call before text is deleted from the buffer; at most one is supported.

int Fl_Text_Buffer::mTabDist [protected]

equiv.
number of characters in a tab

void(∗ Fl_Text_Buffer::transcoding_warning_action)(Fl_Text_Buffer ∗)

Pointer to a function called after reading a non UTF-8 encoded file.

This function is called after reading a file if the file content was transcoded to UTF-8. Its default implementation calls fl_alert() with the text of file_encoding_warning_message. No warning message is displayed if this pointer is set to NULL. Use input_file_was_transcoded to be informed if file input required transcoding to UTF-8.

The documentation for this class was generated from the following files:

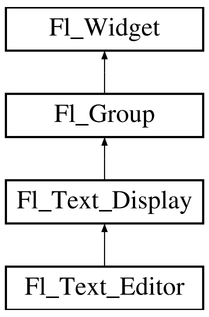
- Fl_Text_Buffer.H
- Fl_Text_Buffer.cxx

31.135 Fl_Text_Display Class Reference

Rich text display widget.

`#include <Fl_Text_Display.H>`

Inheritance diagram for Fl_Text_Display:



Classes

- struct Style_Table_Entry

  This structure associates the color, font, and font size of a string to draw with an attribute mask matching attr.

Public Types

- enum {
  NORMAL_CURSOR, CARET_CURSOR, DIM_CURSOR, BLOCK_CURSOR,
  HEAVY_CURSOR, SIMPLE_CURSOR }

  text display cursor shapes enumeration
- enum { **CURSOR_POS**, **CHARACTER_POS** }

  the character position is the left edge of a character, whereas the cursor is thought to be between the centers of two consecutive characters.
- enum {
  DRAG_NONE = -2, **DRAG_START_DND** = -1, **DRAG_CHAR** = 0, **DRAG_WORD** = 1,
  DRAG_LINE = 2 }

  drag types - they match Fl::event_clicks() so that single clicking to start a collection selects by character, double clicking selects by word and triple clicking selects by line.
- enum { WRAP_NONE, WRAP_AT_COLUMN, WRAP_AT_PIXEL, WRAP_AT_BOUNDS }

  wrap types - used in wrap_mode()
- typedef void(∗ **Unfinished_Style_Cb**)(int, void ∗)

Public Member Functions

- void buffer (Fl_Text_Buffer ∗buf)

  Attach a text buffer to display, replacing the current buffer (if any)
- void buffer (Fl_Text_Buffer &buf)

  Sets the current text buffer associated with the text widget.
- Fl_Text_Buffer ∗ buffer () const

  Gets the current text buffer associated with the text widget.
- double col_to_x (double col) const

  Convert a column number into an x pixel position.
- int count_lines (int start, int end, bool start_pos_is_line_start) const

  Count the number of lines between two positions.
- Fl_Color cursor_color () const

  Gets the text cursor color.
- void cursor_color (Fl_Color n)

Sets the text cursor color.

- void cursor_style (int style)

  Sets the text cursor style.
- Fl_Text_Display (int X, int Y, int W, int H, const char ∗l=0)

  Creates a new text display widget.
- virtual int handle (int e)

  Event handling.
- void hide_cursor ()

  Hides the text cursor.
- void highlight_data (Fl_Text_Buffer ∗styleBuffer, const Style_Table_Entry ∗styleTable, int nStyles, char unfinishedStyle, Unfinished_Style_Cb unfinishedHighlightCB, void ∗cbArg)

  Attach (or remove) highlight information in text display and redisplay.
- int in_selection (int x, int y) const

  Check if a pixel position is within the primary selection.
- void insert (const char ∗text)

  Inserts "text" at the current cursor location.
- void insert_position (int newPos)

  Sets the position of the text insertion cursor for text display.
- int insert_position () const

  Gets the position of the text insertion cursor for text display.
- int line_end (int startPos, bool startPosIsLineStart) const

  Returns the end of a line.
- int line_start (int pos) const

  Return the beginning of a line.
- void linenumber_align (Fl_Align val)

  Set alignment for line numbers (if enabled).
- Fl_Align linenumber_align () const

  Returns the alignment used for line numbers (if enabled).
- void linenumber_bgcolor (Fl_Color val)

  Set the background color used for line numbers (if enabled).
- Fl_Color linenumber_bgcolor () const

  Returns the background color used for line numbers (if enabled).
- void linenumber_fgcolor (Fl_Color val)

  Set the foreground color used for line numbers (if enabled).
- Fl_Color linenumber_fgcolor () const

  Return the foreground color used for line numbers (if enabled).
- void linenumber_font (Fl_Font val)

  Set the font used for line numbers (if enabled).
- Fl_Font linenumber_font () const

  Return the font used for line numbers (if enabled).
- void linenumber_format (const char ∗val)

  Sets the printf() style format string used for line numbers.
- const char ∗ linenumber_format () const

  Returns the line number printf() format string.
- void linenumber_size (Fl_Fontsize val)

  Set the font size used for line numbers (if enabled).
- Fl_Fontsize linenumber_size () const

Return the font size used for line numbers (if enabled).

- void linenumber_width (int width)

  Set width of screen area for line numbers.
- int linenumber_width () const

  Return the screen area width provided for line numbers.
- int move_down ()

  Moves the current insert position down one line.
- int move_left ()

  Moves the current insert position left one character.
- int move_right ()

  Moves the current insert position right one character.
- int move_up ()

  Moves the current insert position up one line.
- void next_word (void)

  Moves the current insert position right one word.
- void overstrike (const char ∗text)

  Replaces text at the current insert position.
- int position_style (int lineStartPos, int lineLen, int lineIndex) const

  Find the correct style for a character.
- int position_to_xy (int pos, int ∗x, int ∗y) const

  Convert a character index into a pixel position.
- void previous_word (void)

  Moves the current insert position left one word.
- void redisplay_range (int start, int end)

  Marks text from start to end as needing a redraw.
- virtual void resize (int X, int Y, int W, int H)

  Change the size of the displayed text area.
- int rewind_lines (int startPos, int nLines)

  Skip a number of lines back.
- void scroll (int topLineNum, int horizOffset)

  Scrolls the current buffer to start at the specified line and column.
- Fl_Align scrollbar_align () const

  Gets the scrollbar alignment type.
- void scrollbar_align (Fl_Align a)

  Sets the scrollbar alignment type.
- int scrollbar_width () const

  Gets the width/height of the scrollbars.
- void scrollbar_width (int W)

  Sets the width/height of the scrollbars.
- int shortcut () const
- void shortcut (int s)
- void show_cursor (int b=1)

  Shows the text cursor.
- void show_insert_position ()

  Scrolls the text buffer to show the current insert position.
- int skip_lines (int startPos, int nLines, bool startPosIsLineStart)

  Skip a number of lines forward.

- Fl_Color textcolor () const

  Gets the default color of text in the widget.
- void textcolor (Fl_Color n)

  Sets the default color of text in the widget.
- Fl_Font textfont () const

  Gets the default font used when drawing text in the widget.
- void textfont (Fl_Font s)

  Sets the default font used when drawing text in the widget.
- Fl_Fontsize textsize () const

  Gets the default size of text in the widget.
- void textsize (Fl_Fontsize s)

  Sets the default size of text in the widget.
- int word_end (int pos) const

  Moves the insert position to the end of the current word.
- int word_start (int pos) const

  Moves the insert position to the beginning of the current word.
- void wrap_mode (int wrap, int wrap_margin)

  Set the new text wrap mode.
- int wrapped_column (int row, int column) const

  Nobody knows what this function does.
- int wrapped_row (int row) const

  Nobody knows what this function does.
- double x_to_col (double x) const

  Convert an x pixel position into a column number.
- ∼Fl_Text_Display ()

  Free a text display and release its associated memory.

Protected Types

- enum { **DRAW_LINE**, **FIND_INDEX**, **FIND_INDEX_FROM_ZERO**, **GET_WIDTH** }

Protected Member Functions

- void absolute_top_line_number (int oldFirstChar)

  Line numbering stuff, currently unused.
- void calc_last_char ()

  Update last display character index.
- void calc_line_starts (int startLine, int endLine)

  Update the line start arrays.
- void clear_rect (int style, int x, int y, int width, int height) const

  Clear a rectangle with the appropriate background color for `style`.
- void display_insert ()

  Scroll the display to bring insertion cursor into view.
- virtual void draw ()

  Draw the widget.
- void draw_cursor (int, int)

  Draw a cursor with top center at `X`, `Y`.
- void draw_line_numbers (bool clearAll)

Refresh the line number area.
- void draw_range (int start, int end)

  Draw a range of text.
- void draw_string (int style, int x, int y, int toX, const char ∗string, int nChars) const

  Draw a text segment in a single style.
- void draw_text (int X, int Y, int W, int H)

  Refresh a rectangle of the text display.
- void draw_vline (int visLineNum, int leftClip, int rightClip, int leftCharIndex, int rightCharIndex)

  Draw a single line of text.
- int empty_vlines () const

  Return true if there are lines visible with no corresponding buffer text.
- void extend_range_for_styles (int ∗start, int ∗end)

  I don't know what this does!
- void find_line_end (int pos, bool start_pos_is_line_start, int ∗lineEnd, int ∗nextLineStart) const

  Finds both the end of the current line and the start of the next line.
- void find_wrap_range (const char ∗deletedText, int pos, int nInserted, int nDeleted, int ∗modRange-Start, int ∗modRangeEnd, int ∗linesInserted, int ∗linesDeleted)

  Wrapping calculations.
- int find_x (const char ∗s, int len, int style, int x) const

  Find the index of the character that lies at the given x position.
- int get_absolute_top_line_number () const

  Line numbering stuff, currently unused.
- int handle_vline (int mode, int lineStart, int lineLen, int leftChar, int rightChar, int topClip, int bottomClip, int leftClip, int rightClip) const

  Universal pixel machine.
- int longest_vline () const

  Find the longest line of all visible lines.
- void maintain_absolute_top_line_number (int state)

  Line numbering stuff, currently unused.
- int maintaining_absolute_top_line_number () const

  Line numbering stuff, currently unused.
- void measure_deleted_lines (int pos, int nDeleted)

  Wrapping calculations.
- double measure_proportional_character (const char ∗s, int colNum, int pos) const

  Wrapping calculations.
- int measure_vline (int visLineNum) const

  Returns the width in pixels of the displayed line pointed to by "visLineNum".
- void offset_line_starts (int newTopLineNum)

  Offset line start counters for a new vertical scroll position.
- int position_to_line (int pos, int ∗lineNum) const

  Convert a position index into a line number offset.
- int position_to_linecol (int pos, int ∗lineNum, int ∗column) const

  Find the line and column number of position `pos`.
- void reset_absolute_top_line_number ()

  Line numbering stuff, probably unused.
- int scroll_ (int topLineNum, int horizOffset)

  Scrolls the current buffer to start at the specified line and column.

- double string_width (const char ∗string, int length, int style) const

    Find the width of a string in the font of a particular style.
- void update_h_scrollbar ()

    Update horizontal scrollbar.
- void update_line_starts (int pos, int charsInserted, int charsDeleted, int linesInserted, int linesDeleted, int ∗scrolled)

    Update line start arrays and variables.
- void update_v_scrollbar ()

    Update vertical scrollbar.
- int vline_length (int visLineNum) const

    Count number of bytes in a visible line.
- int wrap_uses_character (int lineEndPos) const

    Check if the line break is caused by a \n or by line wrapping.
- void wrapped_line_counter (Fl_Text_Buffer ∗buf, int startPos, int maxPos, int maxLines, bool startPosIsLineStart, int styleBufOffset, int ∗retPos, int ∗retLines, int ∗retLineStart, int ∗retLineEnd, bool countLastLineMissingNewLine=true) const

    Wrapping calculations.
- int xy_to_position (int x, int y, int PosType=CHARACTER_POS) const

    Translate a pixel position into a character index.
- void xy_to_rowcol (int x, int y, int ∗row, int ∗column, int PosType=CHARACTER_POS) const

    Translate pixel coordinates into row and column.

Static Protected Member Functions

- static void buffer_modified_cb (int pos, int nInserted, int nDeleted, int nRestyled, const char ∗deletedText, void ∗cbArg)

    This is called whenever the buffer is modified.
- static void buffer_predelete_cb (int pos, int nDeleted, void ∗cbArg)

    This is called before any characters are deleted.
- static void h_scrollbar_cb (Fl_Scrollbar ∗w, Fl_Text_Display ∗d)

    Callbacks for drag or valueChanged on horizontal scrollbar.
- static void scroll_timer_cb (void ∗)

    Timer callback for scroll events.
- static void v_scrollbar_cb (Fl_Scrollbar ∗w, Fl_Text_Display ∗d)

    Callbacks for drag or valueChanged on vertical scrollbar.

Protected Attributes

- int **damage_range1_end**
- int **damage_range1_start**
- int **damage_range2_end**
- int **damage_range2_start**
- int **display_insert_position_hint**
- int **dragging**
- int **dragPos**
- int **dragType**
- Fl_Align **linenumber_align_**
- Fl_Color **linenumber_bgcolor_**
- Fl_Color **linenumber_fgcolor_**

- Fl_Font **linenumber_font_**
- const char ∗ **linenumber_format_**
- Fl_Fontsize **linenumber_size_**
- int **mAbsTopLineNum**
- Fl_Text_Buffer ∗ **mBuffer**
- double **mColumnScale**
- int **mContinuousWrap**
- Fl_Color **mCursor_color**
- int **mCursorOldY**
- int **mCursorOn**
- int **mCursorPos**
- int **mCursorPreferredXPos**
- int **mCursorStyle**
- int **mCursorToHint**
- int **mFirstChar**
- void ∗ **mHighlightCBArg**
- int **mHorizOffset**
- int **mHorizOffsetHint**
- Fl_Scrollbar ∗ **mHScrollBar**
- int **mLastChar**
- int **mLineNumLeft**
- int **mLineNumWidth**
- int ∗ **mLineStarts**
- int **mMaxsize**
- int **mModifyingTabDistance**
- int **mNBufferLines**
- int **mNeedAbsTopLineNum**
- int **mNLinesDeleted**
- int **mNStyles**
- int **mNVisibleLines**
- Fl_Text_Buffer ∗ **mStyleBuffer**
- const Style_Table_Entry ∗ **mStyleTable**
- int **mSuppressResync**
- int **mTopLineNum**
- int **mTopLineNumHint**
- Unfinished_Style_Cb **mUnfinishedHighlightCB**
- char **mUnfinishedStyle**
- Fl_Scrollbar ∗ **mVScrollBar**
- int **mWrapMarginPix**
- Fl_Align **scrollbar_align_**
- int **scrollbar_width_**
- int **shortcut_**

- Fl_Color **textcolor_**
  Fl_Font **textfont_**
  Fl_Fontsize **textsize_**

Friends

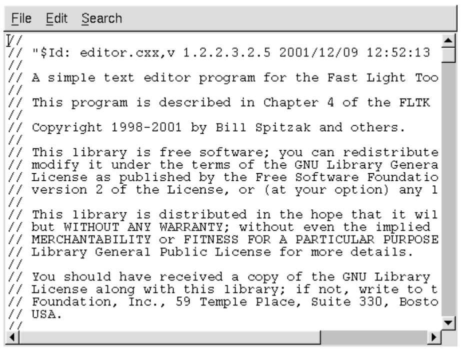
- void **fl_text_drag_me** (int pos, Fl_Text_Display ∗d)

Additional Inherited Members

31.135.1 Detailed Description

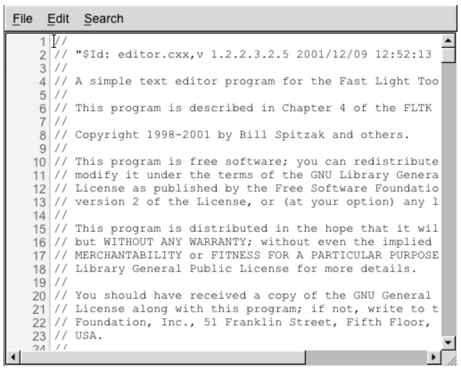
Rich text display widget.

This is the FLTK text display widget. It allows the user to view multiple lines of text and supports highlighting, word wrap, mixes of font faces and colors, line numbers and scrolling. The buffer that is displayed in the widget is managed by the Fl_Text_Buffer class. A single Text Buffer can be displayed by multiple Text Displays.



```
//
// "$Id: editor.cxx,v 1.2.2.3.2.5 2001/12/09 12:52:13
//
// A simple text editor program for the Fast Light Too
//
// This program is described in Chapter 4 of the FLTK
//
// Copyright 1998-2001 by Bill Spitzak and others.
//
// This library is free software; you can redistribute
// modify it under the terms of the GNU Library Genera
// License as published by the Free Software Foundatio
// version 2 of the License, or (at your option) any l
//
// This library is distributed in the hope that it wil
// but WITHOUT ANY WARRANTY; without even the implied
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE
// Library General Public License for more details.
//
// You should have received a copy of the GNU Library
// License along with this library; if not, write to t
// Foundation, Inc., 59 Temple Place, Suite 330, Bosto
// USA.
//
```

Figure 31.41: Fl_Text_Display widget



```
//
// "$Id: editor.cxx,v 1.2.2.3.2.5 2001/12/09 12:52:13
//
// A simple text editor program for the Fast Light Too
//
// This program is described in Chapter 4 of the FLTK
//
// Copyright 1998-2001 by Bill Spitzak and others.
//
// This program is free software; you can redistribute
// modify it under the terms of the GNU Library Genera
// License as published by the Free Software Foundatio
// version 2 of the License, or (at your option) any l
//
// This program is distributed in the hope that it wil
// but WITHOUT ANY WARRANTY; without even the implied
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE
// Library General Public License for more details.
//
// You should have received a copy of the GNU General
// License along with this program; if not, write to t
// Foundation, Inc., 51 Franklin Street, Fifth Floor,
// USA.
//
```

Figure 31.42: Fl_Text_Display widget with line numbers enabled

Example Use

```
#include <FL/FL_Text_Display.H>
..
int main() {
    ..
    Fl_Text_Buffer *buff = new Fl_Text_Buffer();
    Fl_Text_Display *disp = new Fl_Text_Display(10, 10, 640, 480);
    disp->buffer(buff);                 // attach text buffer to display widget
    buff->text("line one\nline two");   // add some text to buffer
    ..
}
```

Features

- Word wrap: wrap_mode(), wrapped_column(), wrapped_row()
- Font control: textfont(), textsize(), textcolor()

- Font styling: highlight_data()
- Cursor: cursor_style(), show_cursor(), hide_cursor(), cursor_color()
- Line numbers: linenumber_width(), linenumber_font(), linenumber_size(), linenumber_fgcolor(), linenumber_bgcolor(), linenumber_align(), linenumber_format()

Note that other features may be available via Fl_Text_Editor and Fl_Text_Buffer classes.

Note

Line numbers were added in 1.3.3. To avoid breaking ABI, many of its options are read only. To adjust these features in 1.3.x, you must build FLTK with FLTK_ABI_VERSION set to 10303 or higher.

31.135.2 Member Enumeration Documentation

anonymous enum

text display cursor shapes enumeration

Enumerator

NORMAL_CURSOR I-beam.

CARET_CURSOR caret under the text

DIM_CURSOR dim I-beam

BLOCK_CURSOR unfille box under the current character

HEAVY_CURSOR thick I-beam

SIMPLE_CURSOR as cursor as Fl_Input cursor

anonymous enum

wrap types - used in wrap_mode()

Enumerator

WRAP_NONE don't wrap text at all

WRAP_AT_COLUMN wrap text at the given text column

WRAP_AT_PIXEL wrap text at a pixel position

WRAP_AT_BOUNDS wrap text so that it fits into the widget width

31.135.3 Constructor & Destructor Documentation

Fl_Text_Display::Fl_Text_Display (int *X*, int *Y*, int *W*, int *H*, const char ∗ *l* = `0`)

Creates a new text display widget.

Parameters

X,Y,W,H	position and size of widget
l	label text, defaults to none

Fl_Text_Display::∼Fl_Text_Display ( )

Free a text display and release its associated memory.

Note, the text BUFFER that the text display displays is a separate entity and is not freed, nor are the style buffer or style table.

31.135.4 Member Function Documentation

void Fl_Text_Display::absolute_top_line_number (int *oldFirstChar*) `[protected]`

Line numbering stuff, currently unused.

Re-calculate absolute top line number for a change in scroll position.

void Fl_Text_Display::buffer (Fl_Text_Buffer ∗ *buf*)

Attach a text buffer to display, replacing the current buffer (if any)

Parameters

buf	attach this text buffer

void Fl_Text_Display::buffer (Fl_Text_Buffer & *buf*) `[inline]`

Sets the current text buffer associated with the text widget.

Multiple text widgets can be associated with the same text buffer.

Parameters

buf	new text buffer

Fl_Text_Buffer∗ Fl_Text_Display::buffer ( ) const `[inline]`

Gets the current text buffer associated with the text widget.

Multiple text widgets can be associated with the same text buffer.

Returns

current text buffer

void Fl_Text_Display::buffer_modified_cb (int *pos,* int *nInserted,* int *nDeleted,* int *nRestyled,* const char ∗ *deletedText,* void ∗ *cbArg*) `[static], [protected]`

This is called whenever the buffer is modified.

Callback attached to the text buffer to receive modification information

This callback can be used to adjust the display or update other setting. It is not advisable to change any buffers or text in this callback, or line counting may get out of sync.

Parameters

pos	starting index of modification
nInserted	number of bytes we inserted (must be UTF-8 aligned!)
nDeleted	number of bytes deleted (must be UTF-8 aligned!)
nRestyled	??
deletedText	this is what was removed, must not be NULL if nDeleted is set
cbArg	”this” pointer for static callback function

void Fl_Text_Display::buffer_predelete_cb (int *pos,* int *nDeleted,* void ∗ *cbArg*) `[static], [protected]`

This is called before any characters are deleted.

Callback attached to the text buffer to receive delete information before the modifications are actually made.

This callback can be used to adjust the display or update other setting. It is not advisable to change any buffers or text in this callback, or line counting may get out of sync.

Parameters

pos	starting index of deletion
nDeleted	number of bytes we will delete (must be UTF-8 aligned!)
cbArg	"this" pointer for static callback function

void Fl_Text_Display::calc_last_char ( ) `[protected]`

Update last display character index.

Given a Fl_Text_Display with a complete, up-to-date lineStarts array, update the lastChar entry to point to the last buffer position displayed.

void Fl_Text_Display::calc_line_starts (int *startLine*, int *endLine*) `[protected]`

Update the line start arrays.

Scan through the text in the "textD"'s buffer and recalculate the line starts array values beginning at index "startLine" and continuing through (including) "endLine". It assumes that the line starts entry preceding "startLine" (or mFirstChar if startLine is 0) is good, and re-counts newlines to fill in the requested entries. Out of range values for "startLine" and "endLine" are acceptable.

Parameters

startLine,endLine	range of lines to scan as line numbers

void Fl_Text_Display::clear_rect (int *style*, int *X*, int *Y*, int *width*, int *height*) const `[protected]`

Clear a rectangle with the appropriate background color for `style`.

Parameters

style	index into style table
X,Y,width,height	size and position of background area

double Fl_Text_Display::col_to_x (double *col*) const

Convert a column number into an x pixel position.

Parameters

col	an approximate column number based on the main font

Returns

number of pixels from the left margin to the left of an average sized character

int Fl_Text_Display::count_lines (int *startPos*, int *endPos*, bool *startPosIsLineStart*) const

Count the number of lines between two positions.

Same as Fl_Text_Buffer::count_lines(), but takes into account wrapping if wrapping is turned on. If the caller knows that `startPos` is at a line start, it can pass `startPosIsLineStart` as True to make the call more efficient by avoiding the additional step of scanning back to the last newline.

Parameters

startPos	index to first character
endPos	index after last character
startPosIsLine-Start	avoid scanning back to the line start

Returns

number of lines

Fl_Color Fl_Text_Display::cursor_color ( ) const `[inline]`

Gets the text cursor color.

Returns

cursor color

void Fl_Text_Display::cursor_color (Fl_Color *n*) `[inline]`

Sets the text cursor color.
Parameters

n	new cursor color

void Fl_Text_Display::cursor_style (int *style*)

Sets the text cursor style.
Sets the text cursor style to one of the following:

- Fl_Text_Display::NORMAL_CURSOR - Shows an I beam.
- Fl_Text_Display::CARET_CURSOR - Shows a caret under the text.
- Fl_Text_Display::DIM_CURSOR - Shows a dimmed I beam.
- Fl_Text_Display::BLOCK_CURSOR - Shows an unfilled box around the current character.
- Fl_Text_Display::HEAVY_CURSOR - Shows a thick I beam.

This call also switches the cursor on and may trigger a redraw.
Parameters

style	new cursor style

void Fl_Text_Display::display_insert ( ) `[protected]`

Scroll the display to bring insertion cursor into view.
Note: it would be nice to be able to do this without counting lines twice (scroll_() counts them too) and/or to count from the most efficient starting point, but the efficiency of this routine is not as important to the overall performance of the text display.

Todo Unicode?

void Fl_Text_Display::draw (void) `[protected]`, `[virtual]`

Draw the widget.

This function tries to limit drawing to smaller areas if possible.

Reimplemented from Fl_Group.

void Fl_Text_Display::draw_cursor (int *X*, int *Y*) `[protected]`

Draw a cursor with top center at `X`, `Y`.

Parameters

X,Y	cursor position in pixels

void Fl_Text_Display::draw_line_numbers (bool *clearAll*) `[protected]`

Refresh the line number area.
Parameters

clearAll	– (currently unused) If False, only draws the line number text, does not clear the area behind it. If True, clears the area and redraws the text. Use False to avoid a 'flash' for single buffered windows.

void Fl_Text_Display::draw_range (int *startpos*, int *endpos*) `[protected]`

Draw a range of text.

Refresh all of the text between buffer positions `startpos` and `endpos` not including the character at the position `endpos`.

If `endpos` points beyond the end of the buffer, refresh the whole display after `startpos`, including blank lines which are not technically part of any range of characters.
Parameters

startpos	index of first character to draw
endpos	index after last character to draw

void Fl_Text_Display::draw_string (int *style*, int *X*, int *Y*, int *toX*, const char ∗ *string*, int *nChars*) const `[protected]`

Draw a text segment in a single style.

Draw a string or blank area according to parameter `style`, using the appropriate colors and drawing method for that style, with top left corner at `X`, `Y`. If style says to draw text, use `string` as source of characters, and draw `nChars`, if style is FILL, erase rectangle where text would have drawn from `X` to `toX` and from `Y` to the maximum y extent of the current font(s).
Parameters

style	index into style lookup table
X,Y	drawing origin
toX	rightmost position if this is a fill operation
string	text if this is a drawing operation
nChars	number of characters to draw

void Fl_Text_Display::draw_text (int *left*, int *top*, int *width*, int *height*) `[protected]`

Refresh a rectangle of the text display.
Parameters

left,top	are in coordinates of the text drawing window.
width,height	size in pixels

void Fl_Text_Display::draw_vline (int *visLineNum*, int *leftClip*, int *rightClip*, int *leftCharIndex*, int *rightCharIndex*) `[protected]`

Draw a single line of text.

Draw the text on a single line represented by `visLineNum` (the number of lines down from the top of the display), limited by `leftClip` and `rightClip` window coordinates and `leftCharIndex` and `rightCharIndex` character positions (not including the character at position `rightCharIndex`).

Parameters

visLineNum	index of line in the visible line number lookup
leftClip,right-Clip	pixel position of clipped area
leftChar-Index,right-CharIndex	index into line of segment that we want to draw

int Fl_Text_Display::empty_vlines ( ) const `[protected]`

Return true if there are lines visible with no corresponding buffer text.

Returns

1 if there are empty lines

void Fl_Text_Display::extend_range_for_styles (int ∗ *startpos*, int ∗ *endpos*) `[protected]`

I don't know what this does!

Extend the range of a redraw request (from ∗start to ∗end) with additional redraw requests resulting from changes to the attached style buffer (which contains auxiliary information for coloring or styling text).

Parameters

startpos	??
endpos	??

Todo Unicode?

void Fl_Text_Display::find_line_end (int *startPos*, bool *startPosIsLineStart*, int ∗ *lineEnd*, int ∗ *nextLineStart*) const `[protected]`

Finds both the end of the current line and the start of the next line.

Why? In continuous wrap mode, if you need to know both, figuring out one from the other can be expensive or error prone. The problem comes when there's a trailing space or tab just before the end of the buffer. To translate an end of line value to or from the next lines start value, you need to know whether the trailing space or tab is being used as a line break or just a normal character, and to find that out would otherwise require counting all the way back to the beginning of the line.

Parameters

	startPos	
	startPosIsLine-Start	
`out`	*lineEnd*	
`out`	*nextLineStart*	

void Fl_Text_Display::find_wrap_range (const char ∗ *deletedText,* int *pos,* int *nInserted,* int *nDeleted,* int ∗ *modRangeStart,* int ∗ *modRangeEnd,* int ∗ *linesInserted,* int ∗ *linesDeleted*) `[protected]`

Wrapping calculations.

When continuous wrap is on, and the user inserts or deletes characters, wrapping can happen before and beyond the changed position. This routine finds the extent of the changes, and counts the deleted and inserted lines over that range. It also attempts to minimize the size of the range to what has to be counted and re-displayed, so the results can be useful both for delimiting where the line starts need to be recalculated, and for deciding what part of the text to redisplay.

Parameters

deletedText	
pos	
nInserted	
nDeleted	
modRangeStart	
modRangeEnd	
linesInserted	
linesDeleted	

int Fl_Text_Display::find_x (const char ∗ *s,* int *len,* int *style,* int *x*) const `[protected]`

Find the index of the character that lies at the given x position.

Parameters

s	UTF-8 text string
len	length of string
style	index into style lookup table
x	position in pixels

Returns

index into buffer

int Fl_Text_Display::get_absolute_top_line_number ( ) const `[protected]`

Line numbering stuff, currently unused.

Returns the absolute (non-wrapped) line number of the first line displayed. Returns 0 if the absolute top line number is not being maintained.

int Fl_Text_Display::handle_vline (int *mode,* int *lineStartPos,* int *lineLen,* int *leftChar,* int *rightChar,* int *Y,* int *bottomClip,* int *leftClip,* int *rightClip*) const `[protected]`

Universal pixel machine.

We use a single function that handles all line layout, measuring, and drawing

- draw a text range
- return the width of a text range in pixels
- return the index of a character that is at a pixel position

Parameters

in	*mode*	DRAW_LINE, GET_WIDTH, FIND_INDEX
in	*lineStartPos*	index of first character
in	*lineLen*	size of string in bytes
in	*leftChar,right-Char*	
in	*Y*	drawing position
in	*bottomClip,left-Clip,rightClip*	stop work when we reach the clipped area. rightClip is the X position that we search in FIND_INDEX.

Return values

DRAW_LINE	index of last drawn character
GET_WIDTH	width in pixels of text segment if we would draw it
FIND_INDEX	index of character at given x position in window coordinates
FIND_INDEX_FROM_Z-ERO	index of character at given x position without scrolling and widget offsets

Todo we need to handle hidden hyphens and tabs here!

we handle all styles and selections

we must provide code to get pixel positions of the middle of a character as well

void Fl_Text_Display::highlight_data (Fl_Text_Buffer ∗ *styleBuffer*, const Style_Table_Entry ∗ *styleTable*, int *nStyles*, char *unfinishedStyle*, Unfinished_Style_Cb *unfinishedHighlightCB*, void ∗ *cbArg*)

Attach (or remove) highlight information in text display and redisplay.

Highlighting information consists of a style buffer which parallels the normal text buffer, but codes font and color information for the display; a style table which translates style buffer codes (indexed by buffer character - 'A') into fonts and colors; and a callback mechanism for as-needed highlighting, triggered by a style buffer entry of "unfinishedStyle". Style buffer can trigger additional redisplay during a normal buffer modification if the buffer contains a primary Fl_Text_Selection (see extendRangeForStyleMods for more information on this protocol).

Style buffers, tables and their associated memory are managed by the caller.

Styles are ranged from 65 ('A') to 126.

Parameters

styleBuffer	this buffer works in parallel to the text buffer. For every character in the text buffer, the stye buffer has a byte at the same offset that contains an index into an array of possible styles.
styleTable	a list of styles indexed by the style buffer
nStyles	number of styles in the style table
unfinishedStyle	if this style is found, the callback below is called
unfinished-HighlightCB	if a character with an unfinished style is found, this callback will be called
cbArg	and optional argument for the callback above, usually a pointer to the Text Display.

int Fl_Text_Display::in_selection (int *X*, int *Y*) const

Check if a pixel position is within the primary selection.

Parameters

X,Y	pixel position to test

Returns

1 if position (X, Y) is inside of the primary Fl_Text_Selection

void Fl_Text_Display::insert (const char ∗ *text*)

Inserts ”text” at the current cursor location.

This has the same effect as inserting the text into the buffer using BufInsert and then moving the insert position after the newly inserted text, except that it's optimized to do less redrawing.

Parameters

text	new text in UTF-8 encoding.

void Fl_Text_Display::insert_position (int *newPos*)

Sets the position of the text insertion cursor for text display.

Move the insertion cursor in front of the character at `newPos`. This function may trigger a redraw.

Parameters

newPos	new caret position

int Fl_Text_Display::insert_position ( ) const `[inline]`

Gets the position of the text insertion cursor for text display.

Returns

insert position index into text buffer

int Fl_Text_Display::line_end (int *startPos,* bool *startPosIsLineStart*) const

Returns the end of a line.

Same as BufEndOfLine, but takes into account line breaks when wrapping is turned on. If the caller knows that `startPos` is at a line start, it can pass ”startPosIsLineStart” as True to make the call more efficient by avoiding the additional step of scanning back to the last newline.

Note that the definition of the end of a line is less clear when continuous wrap is on. With continuous wrap off, it's just a pointer to the newline that ends the line. When it's on, it's the character beyond the last **displayable** character on the line, where a whitespace character which has been ”converted” to a newline for wrapping is not considered displayable. Also note that a line can be wrapped at a non-whitespace character if the line had no whitespace. In this case, this routine returns a pointer to the start of the next line. This is also consistent with the model used by visLineLength.

Parameters

startPos	index to starting character
startPosIsLine-Start	avoid scanning back to the line start

Returns

new position as index

int Fl_Text_Display::line_start (int *pos*) const

Return the beginning of a line.

Same as BufStartOfLine, but returns the character after last wrap point rather than the last newline.

Parameters

pos	index to starting character

Returns

new position as index

void Fl_Text_Display::linenumber_align (Fl_Align *val*)

Set alignment for line numbers (if enabled).

Valid values are FL_ALIGN_LEFT, FL_ALIGN_CENTER or FL_ALIGN_RIGHT.

Version

1.3.3 ABI feature (ignored in 1.3.x unless FLTK_ABI_VERSION is 10303 or higher)

void Fl_Text_Display::linenumber_bgcolor (Fl_Color *val*)

Set the background color used for line numbers (if enabled).

Version

1.3.3 ABI feature (ignored in 1.3.x unless FLTK_ABI_VERSION is 10303 or higher)

void Fl_Text_Display::linenumber_fgcolor (Fl_Color *val*)

Set the foreground color used for line numbers (if enabled).

Version

1.3.3 ABI feature (ignored in 1.3.x unless FLTK_ABI_VERSION is 10303 or higher)

void Fl_Text_Display::linenumber_font (Fl_Font *val*)

Set the font used for line numbers (if enabled).

Version

1.3.3 ABI feature (ignored in 1.3.x unless FLTK_ABI_VERSION is 10303 or higher)

void Fl_Text_Display::linenumber_format (const char ∗ *val*)

Sets the printf() style format string used for line numbers.

Default is ”%d” for normal unpadded decimal integers.

An internal copy of `val` is allocated and managed; it is automatically freed whenever a new value is assigned, or when the widget is destroyed.

The value of `val` must *not* be NULL.

Example values:

```
- "%d"   -- For normal line numbers without padding (Default)
- "%03d" -- For 000 padding
- "%x"   -- For hexadecimal line numbers
- "%o"   -- For octal line numbers
```

Version

1.3.3 ABI feature (ignored in 1.3.x unless FLTK_ABI_VERSION is 10303 or higher)

void Fl_Text_Display::linenumber_size (Fl_Fontsize *val*)

Set the font size used for line numbers (if enabled).

Version

1.3.3 ABI feature (ignored in 1.3.x unless FLTK_ABI_VERSION is 10303 or higher)

void Fl_Text_Display::linenumber_width (int *width*)

Set width of screen area for line numbers.

Use to also enable/disable line numbers. A value of 0 disables line numbering, values >0 enable the line number display.

Parameters

width	The new width of the area for line numbers to appear, in pixels. 0 disables line numbers (default)

int Fl_Text_Display::longest_vline ( ) const `[protected]`

Find the longest line of all visible lines.

Returns

the width of the longest visible line in pixels

void Fl_Text_Display::maintain_absolute_top_line_number (int *state*) `[protected]`

Line numbering stuff, currently unused.

In continuous wrap mode, internal line numbers are calculated after wrapping. A separate non-wrapped line count is maintained when line numbering is turned on. There is some performance cost to maintaining this line count, so normally absolute line numbers are not tracked if line numbering is off. This routine allows callers to specify that they still want this line count maintained (for use via TextDPosToLineAnd-Col). More specifically, this allows the line number reported in the statistics line to be calibrated in absolute lines, rather than post-wrapped lines.

int Fl_Text_Display::maintaining_absolute_top_line_number ( ) const `[protected]`

Line numbering stuff, currently unused.

Return true if a separate absolute top line number is being maintained (for displaying line numbers or showing in the statistics line).

void Fl_Text_Display::measure_deleted_lines (int *pos,* int *nDeleted*) `[protected]`

Wrapping calculations.

This is a stripped-down version of the findWrapRange() function above, intended to be used to calculate the number of "deleted" lines during a buffer modification. It is called *before* the modification takes place.

This function should only be called in continuous wrap mode with a non-fixed font width. In that case, it is impossible to calculate the number of deleted lines, because the necessary style information is no longer available *after* the modification. In other cases, we can still perform the calculation afterwards (possibly even more efficiently).

Parameters

pos	
nDeleted	

double Fl_Text_Display::measure_proportional_character (const char ∗ *s*, int *xPix*, int *pos*) const `[protected]`

Wrapping calculations.

Measure the width in pixels of the first character of string ”s” at a particular column ”colNum” and buffer position ”pos”. This is for measuring characters in proportional or mixed-width highlighting fonts.

A note about proportional and mixed-width fonts: the mixed width and proportional font code in nedit does not get much use in general editing, because nedit doesn’t allow per-language-mode fonts, and editing programs in a proportional font is usually a bad idea, so very few users would choose a proportional font as a default. There are still probably mixed- width syntax highlighting cases where things don’t redraw properly for insertion/deletion, though static display and wrapping and resizing should now be solid because they are now used for online help display.

Parameters

s	text string
xPix	x pixel position needed for calculating tab widths
pos	offset within string

Returns

width of character in pixels

int Fl_Text_Display::measure_vline (int *visLineNum*) const `[protected]`

Returns the width in pixels of the displayed line pointed to by ”visLineNum”.

Parameters

visLineNum	index into visible lines array

Returns

width of line in pixels

int Fl_Text_Display::move_down ( )

Moves the current insert position down one line.

Returns

1 if the cursor moved, 0 if the beginning of the text was reached

int Fl_Text_Display::move_left ( )

Moves the current insert position left one character.

Returns

1 if the cursor moved, 0 if the beginning of the text was reached

int Fl_Text_Display::move_right ( )

Moves the current insert position right one character.

Returns

1 if the cursor moved, 0 if the end of the text was reached

int Fl_Text_Display::move_up ( )

Moves the current insert position up one line.

Returns

1 if the cursor moved, 0 if the beginning of the text was reached

void Fl_Text_Display::offset_line_starts (int *newTopLineNum*) `[protected]`

Offset line start counters for a new vertical scroll position.

Offset the line starts array, mTopLineNum, mFirstChar and lastChar, for a new vertical scroll position given by newTopLineNum. If any currently displayed lines will still be visible, salvage the line starts values, otherwise, count lines from the nearest known line start (start or end of buffer, or the closest value in the mLineStarts array)

Parameters

newTopLine-Num	index into buffer

void Fl_Text_Display::overstrike (const char ∗ *text*)

Replaces text at the current insert position.

Parameters

text	new text in UTF-8 encoding

Todo Unicode? Find out exactly what we do here and simplify.

int Fl_Text_Display::position_style (int *lineStartPos,* int *lineLen,* int *lineIndex*) const

Find the correct style for a character.

Determine the drawing method to use to draw a specific character from "buf". `lineStartPos` gives the character index where the line begins, `lineIndex`, the number of characters past the beginning of the line, and `lineIndex` the number of displayed characters past the beginning of the line. Passing `lineStartPos` of -1 returns the drawing style for "no text".

Why not just: position_style(pos)? Because style applies to blank areas of the window beyond the text boundaries, and because this routine must also decide whether a position is inside of a rectangular Fl_Text_Selection, and do so efficiently, without re-counting character positions from the start of the line.

Note that style is a somewhat incorrect name, drawing method would be more appropriate.

Parameters

lineStartPos	beginning of this line
lineLen	number of bytes in line
lineIndex	position of character within line

Returns

style for the given character

int Fl_Text_Display::position_to_line (int *pos*, int ∗ *lineNum*) const `[protected]`

Convert a position index into a line number offset.

Find the line number of position `pos` relative to the first line of displayed text. Returns 0 if the line is not displayed.

Parameters

	pos	??
`out`	*lineNum*	??

Returns

??

Todo What does this do?

int Fl_Text_Display::position_to_linecol (int *pos*, int ∗ *lineNum*, int ∗ *column*) const `[protected]`

Find the line and column number of position `pos`.

This only works for displayed lines. If the line is not displayed, the function returns 0 (without the mLineStarts array it could turn in to very long calculation involving scanning large amounts of text in the buffer). If continuous wrap mode is on, returns the absolute line number (as opposed to the wrapped line number which is used for scrolling).

Parameters

	pos	character index
`out`	*lineNum*	absolute (unwrapped) line number
`out`	*column*	character offset to the beginning of the line

Returns

0 if `pos` is off screen, line number otherwise

Todo a column number makes little sense in the UTF-8/variable font width environment. We will have to further define what exactly we want to return. Please check the functions that call this particular function.

int Fl_Text_Display::position_to_xy (int *pos*, int ∗ *X*, int ∗ *Y*) const

Convert a character index into a pixel position.

Translate a buffer text position to the XY location where the top left of the cursor would be positioned to point to that character. Returns 0 if the position is not displayed because it is ***vertically*** *out* of view. If the position is horizontally out of view, returns the X coordinate where the position would be if it were visible.

Parameters

	pos	character index
out	*X,Y*	pixel position of character on screen

Returns

0 if character vertically out of view, X & Y positions otherwise

void Fl_Text_Display::redisplay_range (int *startpos*, int *endpos*)

Marks text from start to end as needing a redraw.
This function will trigger a damage event and later a redraw of parts of the widget.
Parameters

startpos	index of first character needing redraw
endpos	index after last character needing redraw

void Fl_Text_Display::reset_absolute_top_line_number ( ) `[protected]`

Line numbering stuff, probably unused.
Count lines from the beginning of the buffer to reestablish the absolute (non-wrapped) top line number. If mode is not continuous wrap, or the number is not being maintained, does nothing.

void Fl_Text_Display::resize (int *X*, int *Y*, int *W*, int *H*) `[virtual]`

Change the size of the displayed text area.
Calling this function will trigger a recalculation of all lines visible and of all scrollbar sizes.
Parameters

X,Y,W,H	new position and size of this widget

Reimplemented from Fl_Group.

int Fl_Text_Display::rewind_lines (int *startPos*, int *nLines*)

Skip a number of lines back.
Same as BufCountBackwardNLines, but takes into account line breaks when wrapping is turned on.
Parameters

startPos	index to starting character
nLines	number of lines to skip back

Returns

new position as index

void Fl_Text_Display::scroll (int *topLineNum*, int *horizOffset*)

Scrolls the current buffer to start at the specified line and column.
Parameters

topLineNum	top line number
horizOffset	column number

Todo Column numbers make little sense here.

int Fl_Text_Display::scroll_ (int *topLineNum*, int *horizOffset*) `[protected]`

Scrolls the current buffer to start at the specified line and column.
Parameters

topLineNum	top line number
horizOffset	in pixels

Returns

0 if nothing changed, 1 if we scrolled

void Fl_Text_Display::scroll_timer_cb (void * *user_data*) `[static]`, `[protected]`

Timer callback for scroll events.
This timer event scrolls the text view proportionally to how far the mouse pointer has left the text area. This allows for smooth scrolling without "wiggeling" the mouse.

Fl_Align Fl_Text_Display::scrollbar_align ( ) const `[inline]`

Gets the scrollbar alignment type.

Returns

scrollbar alignment

void Fl_Text_Display::scrollbar_align (Fl_Align *a*) `[inline]`

Sets the scrollbar alignment type.
Parameters

a	new scrollbar alignment

int Fl_Text_Display::scrollbar_width ( ) const `[inline]`

Gets the width/height of the scrollbars.

Returns

width of scrollbars

void Fl_Text_Display::scrollbar_width (int *W*) `[inline]`

Sets the width/height of the scrollbars.

Parameters

W	width of scrollbars

int Fl_Text_Display::shortcut ( ) const `[inline]`

Todo FIXME : get set methods pointing on shortcut_ have no effects as shortcut_ is unused in this class and derived!

Returns

the current shortcut key

void Fl_Text_Display::shortcut (int *s*) `[inline]`

Todo FIXME : get set methods pointing on shortcut_ have no effects as shortcut_ is unused in this class and derived!

Parameters

s	the new shortcut key

void Fl_Text_Display::show_cursor (int *b* = `1`)

Shows the text cursor.

This function may trigger a redraw.

Parameters

b	show(1) or hide(0) the text cursor (caret).

void Fl_Text_Display::show_insert_position ( )

Scrolls the text buffer to show the current insert position.

This function triggers a complete recalculation, ending in a call to Fl_Text_Display::display_insert()

int Fl_Text_Display::skip_lines (int *startPos*, int *nLines*, bool *startPosIsLineStart*)

Skip a number of lines forward.

Same as BufCountForwardNLines, but takes into account line breaks when wrapping is turned on. If the caller knows that startPos is at a line start, it can pass "startPosIsLineStart" as True to make the call more efficient by avoiding the additional step of scanning back to the last newline.

Parameters

startPos	index to starting character
nLines	number of lines to skip ahead
startPosIsLine-Start	avoid scanning back to the line start

Returns

new position as index

double Fl_Text_Display::string_width (const char ∗ *string*, int *length*, int *style*) const `[protected]`

Find the width of a string in the font of a particular style.

Parameters

string	the text
length	number of bytes in string
style	index into style table

Returns

width of text segment in pixels

Fl_Color Fl_Text_Display::textcolor ( ) const `[inline]`

Gets the default color of text in the widget.

Returns

text color unless overridden by a style

void Fl_Text_Display::textcolor (Fl_Color *n*) `[inline]`

Sets the default color of text in the widget.
Parameters

n	new text color

Fl_Font Fl_Text_Display::textfont ( ) const `[inline]`

Gets the default font used when drawing text in the widget.

Returns

current text font face unless overridden by a style

void Fl_Text_Display::textfont (Fl_Font *s*) `[inline]`

Sets the default font used when drawing text in the widget.
Parameters

s	default text font face

Fl_Fontsize Fl_Text_Display::textsize ( ) const `[inline]`

Gets the default size of text in the widget.

Returns

current text height unless overridden by a style

void Fl_Text_Display::textsize (Fl_Fontsize *s*) `[inline]`

Sets the default size of text in the widget.

Parameters

s	new text size

void Fl_Text_Display::update_h_scrollbar ( ) `[protected]`

Update horizontal scrollbar.

Update the minimum, maximum, slider size, page increment, and value for the horizontal scrollbar.

void Fl_Text_Display::update_line_starts (int *pos,* int *charsInserted,* int *charsDeleted,* int *linesInserted,* int *linesDeleted,* int ∗ *scrolled*) `[protected]`

Update line start arrays and variables.

Update the line starts array, mTopLineNum, mFirstChar and lastChar for this text display after a modification to the text buffer, given by the position `pos` where the change began, and the numbers of characters and lines inserted and deleted.

Parameters

	pos	index into buffer of recent changes
	charsInserted	number of bytes(!) inserted
	charsDeleted	number of bytes(!) deleted
	linesInserted	number of lines
	linesDeleted	number of lines
`out`	*scrolled*	set to 1 if the text display needs to be scrolled

void Fl_Text_Display::update_v_scrollbar ( ) `[protected]`

Update vertical scrollbar.

Update the minimum, maximum, slider size, page increment, and value for the vertical scrollbar.

int Fl_Text_Display::vline_length (int *visLineNum*) const `[protected]`

Count number of bytes in a visible line.

Return the length of a line (number of bytes) by examining entries in the line starts array rather than by scanning for newlines.

Parameters

visLineNum	index of line in visible line array

Returns

number of bytes in this line

int Fl_Text_Display::word_end (int *pos*) const `[inline]`

Moves the insert position to the end of the current word.

Parameters

pos	start calculation at this index

Returns

index of first character after the end of the word

int Fl_Text_Display::word_start (int *pos*) const `[inline]`

Moves the insert position to the beginning of the current word.

Parameters

pos	start calculation at this index

Returns

beginning of the words

void Fl_Text_Display::wrap_mode (int *wrap*, int *wrapMargin*)

Set the new text wrap mode.

If `wrap` mode is not zero, this call enables automatic word wrapping at column `wrapMargin`. Word-wrapping does not change the text buffer itself, only the way the text is displayed. Different Text Displays can have different wrap modes, even if they share the same Text Buffer.

Parameters

wrap	new wrap mode is WRAP_NONE (don't wrap text at all), WRAP_AT_COLUMN (wrap text at the given text column), WRAP_AT_PIXEL (wrap text at a pixel position), or WRAP_AT_BOUNDS (wrap text so that it fits into the widget width)
wrapMargin	in WRAP_AT_COLUMN mode, text will wrap at the n'th character. For variable width fonts, an average character width is calculated. The column width is calculated using the current textfont or the first style when this function is called. If the font size changes, this function must be called again. In WRAP_AT_PIXEL mode, this is the pixel position.

Todo we need new wrap modes to wrap at the window edge and based on pixel width or average character width.

int Fl_Text_Display::wrap_uses_character (int *lineEndPos*) const `[protected]`

Check if the line break is caused by a \n or by line wrapping.

Line breaks in continuous wrap mode usually happen at newlines or whitespace. This line-terminating character is not included in line width measurements and has a special status as a non-visible character. However, lines with no whitespace are wrapped without the benefit of a line terminating character, and this distinction causes endless trouble with all of the text display code which was originally written without continuous wrap mode and always expects to wrap at a newline character.

Given the position of the end of the line, as returned by TextDEndOfLine or BufEndOfLine, this returns true if there is a line terminating character, and false if there's not. On the last character in the buffer, this function can't tell for certain whether a trailing space was used as a wrap point, and just guesses that it wasn't. So if an exact accounting is necessary, don't use this function.

Parameters

lineEndPos	index of character where the line wraps

Returns

1 if a \n character causes the line wrap

int Fl_Text_Display::wrapped_column (int *row*, int *column*) const

Nobody knows what this function does.

Correct a column number based on an unconstrained position (as returned by TextDXYToUnconstrained-Position) to be relative to the last actual newline in the buffer before the row and column position given, rather than the last line start created by line wrapping. This is an adapter for rectangular selections and code written before continuous wrap mode, which thinks that the unconstrained column is the number of characters from the last newline. Obviously this is time consuming, because it invloves character re-counting.

Parameters

row	
column	

Returns

something unknown

Todo What does this do and how is it useful? Column numbers mean little in this context. Which functions depend on this one?

Todo Unicode?

void Fl_Text_Display::wrapped_line_counter (Fl_Text_Buffer ∗ *buf,* int *startPos,* int *maxPos,* int *maxLines,* bool *startPosIsLineStart,* int *styleBufOffset,* int ∗ *retPos,* int ∗ *retLines,* int ∗ *retLineStart,* int ∗ *retLineEnd,* bool *countLastLineMissingNewLine* = `true`) const `[protected]`

Wrapping calculations.

Count forward from startPos to either maxPos or maxLines (whichever is reached first), and return all relevant positions and line count. The provided textBuffer may differ from the actual text buffer of the widget. In that case it must be a (partial) copy of the actual text buffer and the styleBufOffset argument must indicate the starting position of the copy, to take into account the correct style information.

Parameters

`in`	*buf*	The text buffer to operate on
`in`	*startPos*	Starting index position into the buffer
`in`	*maxPos*	Maximum index position into the buffer we'll reach
`in`	*maxLines*	Maximum number of lines we'll reach
`in`	*startPosIsLine-Start*	Flag indicating if startPos is start of line. (If set, prevents our having to find the line start)
`in`	*styleBufOffset*	Offset index position into style buffer.
`out`	*retPos*	Position where counting ended. When counting lines, the position returned is the start of the line "maxLines" lines beyond "startPos".
`out`	*retLines*	Number of line breaks counted
`out`	*retLineStart*	Start of the line where counting ended
`out`	*retLineEnd*	End position of the last line traversed
`out`	*countLastLine-MissingNew-Line*	

int Fl_Text_Display::wrapped_row (int *row*) const

Nobody knows what this function does.

Correct a row number from an unconstrained position (as returned by TextDXYToUnconstrained-Position) to a straight number of newlines from the top line of the display. Because rectangular selections are based on newlines, rather than display wrapping, and anywhere a rectangular selection needs a row, it needs it in terms of un-wrapped lines.

Parameters

row	

Returns

something unknown

Todo What does this do and how is it useful? Column numbers mean little in this context. Which functions depend on this one?

double Fl_Text_Display::x_to_col (double *x*) const

Convert an x pixel position into a column number.
Parameters

x	number of pixels from the left margin

Returns

an approximate column number based on the main font

int Fl_Text_Display::xy_to_position (int *X*, int *Y*, int *posType* = *CHARACTER_POS*) const `[protected]`

Translate a pixel position into a character index.
Translate window coordinates to the nearest (insert cursor or character cell) text position. The parameter `posType` specifies how to interpret the position: CURSOR_POS means translate the coordinates to the nearest cursor position, and CHARACTER_POS means return the position of the character closest to (`X`, `Y`).
Parameters

X,Y	pixel position
posType	CURSOR_POS or CHARACTER_POS

Returns

index into text buffer

void Fl_Text_Display::xy_to_rowcol (int *X*, int *Y*, int * *row*, int * *column*, int *posType* = *CHARACTER_POS*) const `[protected]`

Translate pixel coordinates into row and column.
Translate window coordinates to the nearest row and column number for positioning the cursor. This, of course, makes no sense when the font is proportional, since there are no absolute columns. The parameter posType specifies how to interpret the position: CURSOR_POS means translate the coordinates to the nearest position between characters, and CHARACTER_POS means translate the position to the nearest character cell.
Parameters

	X,Y	pixel coordinates

out	*row,column*	neares row and column
	posType	CURSOR_POS or CHARACTER_POS

The documentation for this class was generated from the following files:

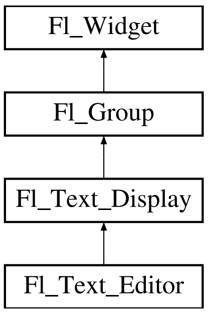
- Fl_Text_Display.H
- Fl_Text_Display.cxx

31.136 Fl_Text_Editor Class Reference

This is the FLTK text editor widget.

`#include <Fl_Text_Editor.H>`

Inheritance diagram for Fl_Text_Editor:



Classes

- struct Key_Binding

  Simple linked list item associating a key/state to a function.

Public Types

- typedef int(∗ Key_Func)(int key, Fl_Text_Editor ∗editor)

  Key function binding callback type.

Public Member Functions

- void add_default_key_bindings (Key_Binding ∗∗list)

  Adds all of the default editor key bindings to the specified key binding list.
- void add_key_binding (int key, int state, Key_Func f, Key_Binding ∗∗list)

  Adds a `key` of state `state` with the function `function` to an arbitrary key binding list `list`.
- void add_key_binding (int key, int state, Key_Func f)

  Adds a `key` of state `state` with the function `f`.
- Key_Func bound_key_function (int key, int state, Key_Binding ∗list) const

  Returns the function associated with a key binding.
- Key_Func bound_key_function (int key, int state) const

  Returns the function associated with a key binding.
- void default_key_function (Key_Func f)

  Sets the default key function for unassigned keys.
- Fl_Text_Editor (int X, int Y, int W, int H, const char ∗l=0)

  The constructor creates a new text editor widget.

- virtual int handle (int e)

  Event handling.
- void insert_mode (int b)

  Sets the current insert mode; if non-zero, new text is inserted before the current cursor position.
- int insert_mode ()

  Gets the current insert mode; if non-zero, new text is inserted before the current cursor position.
- void remove_all_key_bindings (Key_Binding **list)

  Removes all of the key bindings associated with the text editor or list.
- void remove_all_key_bindings ()

  Removes all of the key bindings associated with the text editor or list.
- void remove_key_binding (int key, int state, Key_Binding **list)

  Removes the key binding associated with the key `key` of state `state` from the Key_Binding list `list`.
- void remove_key_binding (int key, int state)

  Removes the key binding associated with the key "key" of state "state".
- void tab_nav (int val)

  Enables or disables Tab key focus navigation.
- int tab_nav () const

  Check if Tab focus navigation is enabled.

Static Public Member Functions

- static int kf_backspace (int c, Fl_Text_Editor *e)

  Does a backspace for key `'c'` in the current buffer of editor `'e'`.
- static int kf_c_s_move (int c, Fl_Text_Editor *e)

  Extends the current selection in the direction indicated by control key `'c'` in editor `'e'`.
- static int kf_copy (int c, Fl_Text_Editor *e)

  Does a copy of selected text or the current character in the current buffer of editor `'e'`.
- static int kf_ctrl_move (int c, Fl_Text_Editor *e)

  Moves the current text cursor in the direction indicated by control key `'c'` in editor `'e'`.
- static int kf_cut (int c, Fl_Text_Editor *e)

  Does a cut of selected text in the current buffer of editor `'e'`.
- static int kf_default (int c, Fl_Text_Editor *e)

  Inserts the text associated with key `'c'` in editor `'e'`.
- static int kf_delete (int c, Fl_Text_Editor *e)

  Does a delete of selected text or the current character in the current buffer of editor `'e'`.
- static int kf_down (int c, Fl_Text_Editor *e)

  Moves the text cursor one line down for editor `'e'`.
- static int kf_end (int c, Fl_Text_Editor *e)

  Moves the text cursor to the end of the current line in editor `'e'`.
- static int kf_enter (int c, Fl_Text_Editor *e)

  Inserts a newline for key `'c'` at the current cursor position in editor `'e'`.
- static int kf_home (int, Fl_Text_Editor *e)

  Moves the text cursor to the beginning of the current line in editor `'e'`.
- static int kf_ignore (int c, Fl_Text_Editor *e)

  Ignores the key `'c'` in editor `'e'`.
- static int kf_insert (int c, Fl_Text_Editor *e)

  Toggles the insert mode for editor `'e'`.

- static int kf_left (int c, Fl_Text_Editor ∗e)

  Moves the text cursor one character to the left in editor `'e'`.
- static int kf_m_s_move (int c, Fl_Text_Editor ∗e)

  Extends the current selection in the direction indicated by meta key `'c'` *in editor* `'e'`.
- static int kf_meta_move (int c, Fl_Text_Editor ∗e)

  Moves the current text cursor in the direction indicated by meta key `'c'` *in editor* `'e'`.
- static int kf_move (int c, Fl_Text_Editor ∗e)

  Moves the text cursor in the direction indicated by key `'c'` *in editor* `'e'`.
- static int kf_page_down (int c, Fl_Text_Editor ∗e)

  Moves the text cursor down one page for editor `'e'`.
- static int kf_page_up (int c, Fl_Text_Editor ∗e)

  Moves the text cursor up one page for editor `'e'`.
- static int kf_paste (int c, Fl_Text_Editor ∗e)

  Does a paste of selected text in the current buffer of editor `'e'`.
- static int kf_right (int c, Fl_Text_Editor ∗e)

  Moves the text cursor one character to the right for editor `'e'`.
- static int kf_select_all (int c, Fl_Text_Editor ∗e)

  Selects all text in the current buffer in editor `'e'`.
- static int kf_shift_move (int c, Fl_Text_Editor ∗e)

  Extends the current selection in the direction of key `'c'` *in editor* `'e'`.
- static int kf_undo (int c, Fl_Text_Editor ∗e)

  Undo last edit in the current buffer of editor `'e'`.
- static int kf_up (int c, Fl_Text_Editor ∗e)

  Moves the text cursor one line up for editor `'e'`.

Protected Member Functions

- int handle_key ()

  Handles a key press in the editor.
- void maybe_do_callback ()

  does or does not a callback according to changed() and when() settings

Static Protected Attributes

- static Key_Binding ∗ global_key_bindings

  Global key binding list.

Additional Inherited Members

31.136.1 Detailed Description

This is the FLTK text editor widget.

It allows the user to edit multiple lines of text and supports highlighting and scrolling. The buffer that is displayed in the widget is managed by the Fl_Text_Buffer class.

31.136.2 Member Typedef Documentation

typedef int(∗ Fl_Text_Editor::Key_Func)(int key, Fl_Text_Editor ∗editor)

Key function binding callback type.

31.136.3 Constructor & Destructor Documentation

Fl_Text_Editor::Fl_Text_Editor (int *X*, int *Y*, int *W*, int *H*, const char * *l* = 0)

The constructor creates a new text editor widget.

31.136.4 Member Function Documentation

void Fl_Text_Editor::add_default_key_bindings (Key_Binding ** *list*)

Adds all of the default editor key bindings to the specified key binding list.

void Fl_Text_Editor::add_key_binding (int *key*, int *state*, Key_Func *function*, Key_Binding ** *list*)

Adds a `key` of state `state` with the function `function` to an arbitrary key binding list `list`.

This can be used in derived classes to add global key bindings by using the global (static) Key_Binding list Fl_Text_Editor::global_key_bindings.

void Fl_Text_Editor::add_key_binding (int *key*, int *state*, Key_Func *f*) `[inline]`

Adds a `key` of state `state` with the function `f`.

Fl_Text_Editor::Key_Func Fl_Text_Editor::bound_key_function (int *key*, int *state*, Key_Binding * *list*) const

Returns the function associated with a key binding.

Key_Func Fl_Text_Editor::bound_key_function (int *key*, int *state*) const `[inline]`

Returns the function associated with a key binding.

void Fl_Text_Editor::default_key_function (Key_Func *f*) `[inline]`

Sets the default key function for unassigned keys.

void Fl_Text_Editor::insert_mode (int *b*) `[inline]`

Sets the current insert mode; if non-zero, new text is inserted before the current cursor position.

Otherwise, new text replaces text at the current cursor position.

int Fl_Text_Editor::insert_mode ( ) `[inline]`

Gets the current insert mode; if non-zero, new text is inserted before the current cursor position.

Otherwise, new text replaces text at the current cursor position.

int Fl_Text_Editor::kf_backspace (int *c*, Fl_Text_Editor * *e*) `[static]`

Does a backspace for key `'c'` in the current buffer of editor `'e'`.

Any current selection is deleted. Otherwise, the character left is deleted and the cursor moved. The key value `'c'` is currently unused.

int Fl_Text_Editor::kf_c_s_move (int *c*, Fl_Text_Editor ∗ *e*) `[static]`

Extends the current selection in the direction indicated by control key `'c'` in editor `'e'`.

See Also

kf_ctrl_move().

int Fl_Text_Editor::kf_copy (int *c*, Fl_Text_Editor ∗ *e*) `[static]`

Does a copy of selected text or the current character in the current buffer of editor `'e'`.
The key value `'c'` is currently unused.

int Fl_Text_Editor::kf_ctrl_move (int *c*, Fl_Text_Editor ∗ *e*) `[static]`

Moves the current text cursor in the direction indicated by control key `'c'` in editor `'e'`.
Supported values for 'c' are currently:

```
FL_Home      -- moves the cursor to the beginning of the document
FL_End       -- moves the cursor to the end of the document
FL_Left      -- moves the cursor left one word
FL_Right     -- moves the cursor right one word
FL_Up        -- scrolls up one line, without moving cursor
FL_Down      -- scrolls down one line, without moving cursor
FL_Page_Up   -- moves the cursor to the beginning of the top line on the
      current page
FL_Page_Down -- moves the cursor to the beginning of the last line on the
      current page
```

int Fl_Text_Editor::kf_cut (int *c*, Fl_Text_Editor ∗ *e*) `[static]`

Does a cut of selected text in the current buffer of editor `'e'`.
The key value `'c'` is currently unused.

int Fl_Text_Editor::kf_default (int *c*, Fl_Text_Editor ∗ *e*) `[static]`

Inserts the text associated with key `'c'` in editor `'e'`.
Honors the current selection and insert/overstrike mode.

int Fl_Text_Editor::kf_delete (int *c*, Fl_Text_Editor ∗ *e*) `[static]`

Does a delete of selected text or the current character in the current buffer of editor `'e'`.
The key value `'c'` is currently unused.

int Fl_Text_Editor::kf_down (int *c*, Fl_Text_Editor ∗ *e*) `[static]`

Moves the text cursor one line down for editor `'e'`.
Same as kf_move(FL_Down, e). The key value `'c'` is currently unused.

int Fl_Text_Editor::kf_end (int *c*, Fl_Text_Editor ∗ *e*) `[static]`

Moves the text cursor to the end of the current line in editor `'e'`.
Same as kf_move(FL_End, e). The key value `'c'` is currently unused.

int Fl_Text_Editor::kf_enter (int *c*, Fl_Text_Editor ∗ *e*) `[static]`

Inserts a newline for key `'c'` at the current cursor position in editor `'e'`.
The key value `'c'` is currently unused.

int Fl_Text_Editor::kf_home (int , Fl_Text_Editor * *e*) `[static]`

Moves the text cursor to the beginning of the current line in editor `'e'`.

Same as kf_move(FL_Home, e). The key value `'c'` is currently unused.

int Fl_Text_Editor::kf_ignore (int *c*, Fl_Text_Editor * *e*) `[static]`

Ignores the key `'c'` in editor `'e'`.

This method can be used as a keyboard binding to disable a key that might otherwise be handled or entered as text.

An example would be disabling FL_Escape, so that it isn't added to the buffer when invoked by the user.

int Fl_Text_Editor::kf_insert (int *c*, Fl_Text_Editor * *e*) `[static]`

Toggles the insert mode for editor `'e'`.

The key value `'c'` is currently unused.

int Fl_Text_Editor::kf_left (int *c*, Fl_Text_Editor * *e*) `[static]`

Moves the text cursor one character to the left in editor `'e'`.

Same as kf_move(FL_Left, e). The key value `'c'` is currently unused.

int Fl_Text_Editor::kf_m_s_move (int *c*, Fl_Text_Editor * *e*) `[static]`

Extends the current selection in the direction indicated by meta key `'c'` in editor `'e'`.

See Also

kf_meta_move().

int Fl_Text_Editor::kf_meta_move (int *c*, Fl_Text_Editor * *e*) `[static]`

Moves the current text cursor in the direction indicated by meta key `'c'` in editor `'e'`.

Supported values for 'c' are currently:

```
FL_Up        -- moves cursor to the beginning of the current document
FL_Down      -- moves cursor to the end of the current document
FL_Left      -- moves the cursor to the beginning of the current line
FL_Right     -- moves the cursor to the end of the current line
```

int Fl_Text_Editor::kf_move (int *c*, Fl_Text_Editor * *e*) `[static]`

Moves the text cursor in the direction indicated by key `'c'` in editor `'e'`.

Supported values for 'c' are currently:

```
FL_Home      -- moves the cursor to the beginning of the current line
FL_End       -- moves the cursor to the end of the current line
FL_Left      -- moves the cursor left one character
FL_Right     -- moves the cursor right one character
FL_Up        -- moves the cursor up one line
FL_Down      -- moves the cursor down one line
FL_Page_Up   -- moves the cursor up one page
FL_Page_Down -- moves the cursor down one page
```

int Fl_Text_Editor::kf_page_down (int *c*, Fl_Text_Editor * *e*) `[static]`

Moves the text cursor down one page for editor `'e'`.

Same as kf_move(FL_Page_Down, e). The key value `'c'` is currently unused.

int Fl_Text_Editor::kf_page_up (int *c*, Fl_Text_Editor ∗ *e*) `[static]`

Moves the text cursor up one page for editor `'e'`.
Same as kf_move(FL_Page_Up, e). The key value `'c'` is currently unused.

int Fl_Text_Editor::kf_paste (int *c*, Fl_Text_Editor ∗ *e*) `[static]`

Does a paste of selected text in the current buffer of editor `'e'`.
Any current selection is replaced with the pasted content. The key value `'c'` is currently unused.

int Fl_Text_Editor::kf_right (int *c*, Fl_Text_Editor ∗ *e*) `[static]`

Moves the text cursor one character to the right for editor `'e'`.
Same as kf_move(FL_Right, e). The key value `'c'` is currently unused.

int Fl_Text_Editor::kf_select_all (int *c*, Fl_Text_Editor ∗ *e*) `[static]`

Selects all text in the current buffer in editor `'e'`.
The key value `'c'` is currently unused.

int Fl_Text_Editor::kf_shift_move (int *c*, Fl_Text_Editor ∗ *e*) `[static]`

Extends the current selection in the direction of key `'c'` in editor `'e'`.

See Also

kf_move()

int Fl_Text_Editor::kf_undo (int *c*, Fl_Text_Editor ∗ *e*) `[static]`

Undo last edit in the current buffer of editor `'e'`.
Also deselects previous selection. The key value `'c'` is currently unused.

int Fl_Text_Editor::kf_up (int *c*, Fl_Text_Editor ∗ *e*) `[static]`

Moves the text cursor one line up for editor `'e'`.
Same as kf_move(FL_Up, e). The key value `'c'` is currently unused.

void Fl_Text_Editor::remove_all_key_bindings (Key_Binding ∗∗ *list*)

Removes all of the key bindings associated with the text editor or list.

void Fl_Text_Editor::remove_all_key_bindings ( ) `[inline]`

Removes all of the key bindings associated with the text editor or list.

void Fl_Text_Editor::remove_key_binding (int *key*, int *state*, Key_Binding ∗∗ *list*)

Removes the key binding associated with the key `key` of state `state` from the Key_Binding list `list`.
This can be used in derived classes to remove global key bindings by using the global (static) Key_-Binding list Fl_Text_Editor::global_key_bindings.

void Fl_Text_Editor::remove_key_binding (int *key*, int *state*) `[inline]`

Removes the key binding associated with the key ”key” of state ”state”.

void Fl_Text_Editor::tab_nav (int *val*)

Enables or disables Tab key focus navigation.

When disabled (default), tab characters are inserted into Fl_Text_Editor. Only the mouse can change focus. This behavior is desireable when Fl_Text_Editor is used, e.g. in a source code editor.

When enabled, Tab navigates focus to the next widget, and Shift-Tab navigates focus to the previous widget. This behavior is desireable when Fl_Text_Editor is used e.g. in a database input form.

Currently, this method is implemented as a convenience method that adjusts the key bindings for the Tab key. This implementation detail may change in the future. Know that changing the editor's key bindings for Tab and Shift-Tab may affect tab navigation.

Parameters

`in`	*val*	If `val` is 0, Tab inserts a tab character (default). If `val` is 1, Tab navigates widget focus.

See Also

tab_nav(), Fl::OPTION_ARROW_FOCUS.

Version

1.3.4 ABI feature

int Fl_Text_Editor::tab_nav ( ) const

Check if Tab focus navigation is enabled.

If disabled (default), hitting Tab inserts a tab character into the editor buffer.

If enabled, hitting Tab navigates focus to the next widget, and Shift-Tab navigates focus to the previous widget.

Returns

if Tab inserts tab characters or moves the focus

Return values

0	Tab inserts tab characters (default)
1	Tab navigation is enabled.

See Also

tab_nav(int), Fl::OPTION_ARROW_FOCUS.

Version

1.3.4 ABI feature

31.136.5 Member Data Documentation

Key_Binding∗ Fl_Text_Editor::global_key_bindings `[static]`, `[protected]`

Global key binding list.

Derived classes can add key bindings for all Fl_Text_Editor widgets by adding a Key_Binding to this list.

See Also

add_key_binding(int key, int state, Key_Func f, Key_Binding∗∗ list);

The documentation for this class was generated from the following files:

- Fl_Text_Editor.H
- Fl_Text_Editor.cxx

31.137 Fl_Text_Selection Class Reference

This is an internal class for Fl_Text_Buffer to manage text selections.

`#include <Fl_Text_Buffer.H>`

Public Member Functions

- int end () const

  Return the byte offset to the character after the last selected character.
- int includes (int pos) const

  Return true if position `pos` with indentation `dispIndex` is in the Fl_Text_Selection.
- int position (int ∗start, int ∗end) const

  Return the positions of this selection.
- bool selected () const

  Returns true if any text is selected.
- void selected (bool b)

  Modify the 'selected' flag.
- void set (int start, int end)

  Set the selection range.
- int start () const

  Return the byte offset to the first selected character.
- void update (int pos, int nDeleted, int nInserted)

  Updates a selection after text was modified.

Protected Attributes

- int mEnd

  byte offset to the character after the last selected character
- bool mSelected

  this flag is set if any text is selected
- int mStart

  byte offset to the first selected character

Friends

- class **Fl_Text_Buffer**

31.137.1 Detailed Description

This is an internal class for Fl_Text_Buffer to manage text selections.

This class works correctly with UTF-8 strings assuming that the parameters for all calls are on character boundaries.

31.137.2 Member Function Documentation

int Fl_Text_Selection::end ( ) const `[inline]`

Return the byte offset to the character after the last selected character.

Returns

byte offset

int Fl_Text_Selection::position (int ∗ *start*, int ∗ *end*) const

Return the positions of this selection.

Parameters

start	return byte offset to first selected character
end	return byte offset pointing after last selected character

Returns

true if selected

bool Fl_Text_Selection::selected ( ) const `[inline]`

Returns true if any text is selected.

Returns

a non-zero number if any text has been selected, or 0 if no text is selected.

void Fl_Text_Selection::selected (bool *b*) `[inline]`

Modify the 'selected' flag.
Parameters

b	new flag

void Fl_Text_Selection::set (int *start*, int *end*)

Set the selection range.
Parameters

start	byte offset to first selected character
end	byte offset pointing after last selected character

int Fl_Text_Selection::start ( ) const `[inline]`

Return the byte offset to the first selected character.

Returns

byte offset

void Fl_Text_Selection::update (int *pos*, int *nDeleted*, int *nInserted*)

Updates a selection after text was modified.
Updates an individual selection for changes in the corresponding text
Parameters

pos	byte offset into text buffer at which the change occurred
nDeleted	number of bytes deleted from the buffer
nInserted	number of bytes inserted into the buffer

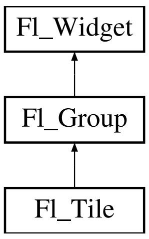
The documentation for this class was generated from the following files:

- Fl_Text_Buffer.H
- Fl_Text_Buffer.cxx

31.138 Fl_Tile Class Reference

The Fl_Tile class lets you resize its children by dragging the border between them.

Inheritance diagram for Fl_Tile:



Public Member Functions

- Fl_Tile (int X, int Y, int W, int H, const char ∗L=0)

  Creates a new Fl_Tile widget using the given position, size, and label string.
- int handle (int event)

  Handles the specified event.
- void position (int oldx, int oldy, int newx, int newy)

  Drags the intersection at (`oldx`,`oldy`) to (`newx`,`newy`).
- void resize (int X, int Y, int W, int H)

  Resizes the Fl_Tile widget and its children.

Additional Inherited Members

31.138.1 Detailed Description

The Fl_Tile class lets you resize its children by dragging the border between them.

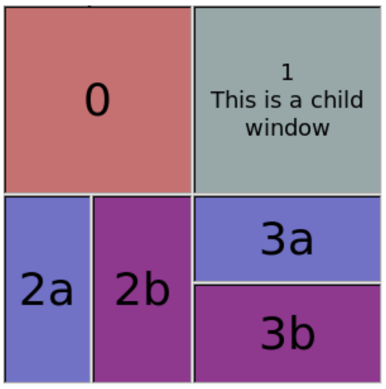


Figure 31.43: Fl_Tile

For the tiling to work correctly, the children of an Fl_Tile must cover the entire area of the widget, but not overlap. This means that all children must touch each other at their edges, and no gaps can be left inside the Fl_Tile.

Fl_Tile does not normally draw any graphics of its own. The "borders" which can be seen in the snapshot above are actually part of the children. Their boxtypes have been set to FL_DOWN_BOX creating the impression of "ridges" where the boxes touch. What you see are actually two adjacent FL_DOWN_BOX's drawn next to each other. All neighboring widgets share the same edge - the widget's thick borders make it appear as though the widgets aren't actually touching, but they are. If the edges of adjacent widgets do not touch, then it will be impossible to drag the corresponding edges.

Fl_Tile allows objects to be resized to zero dimensions. To prevent this you can use the resizable() to limit where corners can be dragged to. For more information see note below.

Even though objects can be resized to zero sizes, they must initially have non-zero sizes so the Fl_Tile can figure out their layout. If desired, call position() after creating the children but before displaying the window to set the borders where you want.

Note on resizable(Fl_Widget &w): The "resizable" child widget (which should be invisible) limits where the borders can be dragged to. All dragging will be limited inside the resizable widget's borders. If you don't set it, it will be possible to drag the borders right to the edges of the Fl_Tile widget, and thus resize objects on the edges to zero width or height. When the entire Fl_Tile widget is resized, the resizable() widget will keep its border distance to all borders the same (this is normal resize behavior), so that you can effectively set a border width that will never change. To ensure correct event delivery to all child widgets the resizable() widget must be the first child of the Fl_Tile widget group. Otherwise some events (e.g. FL_MOVE and FL_ENTER) might be consumed by the resizable() widget so that they are lost for widgets covered (overlapped) by the resizable() widget.

Note

You can still resize widgets **inside** the resizable() to zero width and/or height, i.e. box **2b** above to zero width and box **3a** to zero height.

See Also

void Fl_Group::resizable(Fl_Widget &w)

Example for resizable with 20 pixel border distance:

```
int dx = 20, dy = dx;
Fl_Tile tile(50,50,300,300);
// create resizable() box first
Fl_Box r(tile.x()+dx,tile.y()+dy,tile.w()-2*dx,tile.h()-2*dy);
tile.resizable(r);
// ... create widgets inside tile (see test/tile.cxx) ...
tile.end();
```

See also the complete example program in test/tile.cxx.

31.138.2 Constructor & Destructor Documentation

Fl_Tile::Fl_Tile (int *X*, int *Y*, int *W*, int *H*, const char * *L* = 0)

Creates a new Fl_Tile widget using the given position, size, and label string.

The default boxtype is FL_NO_BOX.

The destructor *also deletes all the children*. This allows a whole tree to be deleted at once, without having to keep a pointer to all the children in the user code. A kludge has been done so the Fl_Tile and all of its children can be automatic (local) variables, but you must declare the Fl_Tile *first*, so that it is destroyed last.

See Also

class Fl_Group

31.138.3 Member Function Documentation

int Fl_Tile::handle (int *event*) `[virtual]`

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited handle() method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters

`in`	*event*	the kind of event received

Return values

0	if the event was not used or understood
1	if the event was used and can be deleted

See Also

Fl_Event

Reimplemented from Fl_Group.

void Fl_Tile::position (int *oldx*, int *oldy*, int *newx*, int *newy*)

Drags the intersection at (`oldx`,`oldy`) to (`newx`,`newy`).

This redraws all the necessary children.

Pass zero as `oldx` or `oldy` to disable drag in that direction.

void Fl_Tile::resize (int *X*, int *Y*, int *W*, int *H*) `[virtual]`

Resizes the Fl_Tile widget and its children.

Fl_Tile implements its own resize() method. It does not use Fl_Group::resize() to resize itself and its children.

Enlarging works by just moving the lower-right corner and resizing the bottom and right border widgets accordingly.

Shrinking the Fl_Tile works in the opposite way by shrinking the bottom and right border widgets, unless they are reduced to zero width or height, resp. or to their minimal sizes defined by the resizable() widget. In this case other widgets will be shrunk as well.

See the Fl_Tile class documentation about how the resizable() works.

Reimplemented from Fl_Group.

The documentation for this class was generated from the following files:

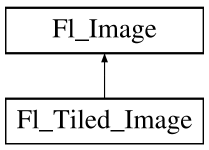
- Fl_Tile.H
- Fl_Tile.cxx

31.139 Fl_Tiled_Image Class Reference

This class supports tiling of images over a specified area.

`#include <Fl_Tiled_Image.H>`

Inheritance diagram for Fl_Tiled_Image:



Public Member Functions

- virtual void color_average (Fl_Color c, float i)

  The color_average() method averages the colors in the image with the FLTK color value c.
- virtual Fl_Image ∗ copy (int W, int H)

  The copy() method creates a copy of the specified image.
- Fl_Image ∗ **copy** ()
- virtual void desaturate ()

  The desaturate() method converts an image to grayscale.
- virtual void draw (int X, int Y, int W, int H, int cx, int cy)

  Draws a tiled image.
- void **draw** (int X, int Y)
- Fl_Tiled_Image (Fl_Image ∗i, int W=0, int H=0)

  The constructors create a new tiled image containing the specified image.
- Fl_Image ∗ image ()

  Gets The image that is tiled.
- virtual ∼Fl_Tiled_Image ()

  The destructor frees all memory and server resources that are used by the tiled image.

Protected Attributes

- int **alloc_image_**
- Fl_Image ∗ **image_**

Additional Inherited Members

31.139.1 Detailed Description

This class supports tiling of images over a specified area.

The source (tile) image is **not** copied unless you call the color_average(), desaturate(), or inactive() methods.

31.139.2 Constructor & Destructor Documentation

Fl_Tiled_Image::Fl_Tiled_Image (Fl_Image ∗ *i*, int *W* = *0*, int *H* = *0*)

The constructors create a new tiled image containing the specified image.

Use a width and height of 0 to tile the whole window/widget.

Note

Due to implementation constraints in FLTK 1.3.3 and later width and height of 0 may not work as expected when used as background image in widgets other than windows. You may need to center and clip the image (label) and set the label type to FL_NORMAL_LABEL. Doing so will let the tiled image fill the whole widget as its background image. Other combinations of label flags may or may not work.

```
#include "bg.xpm"
Fl_Pixmap *bg_xpm = new Fl_Pixmap(bg_xpm);
Fl_Tiled_Image *bg_tiled = new Fl_Tiled_Image(bg_xpm,0,0);

Fl_Box *box = new Fl_Box(40,40,300,100,"");
box->box(FL_UP_BOX);
box->labeltype(FL_NORMAL_LABEL);
box->align(FL_ALIGN_INSIDE | FL_ALIGN_CENTER |
      FL_ALIGN_CLIP);
box->image(bg_tiled);
```

Note

Setting an image (label) for a window may not work as expected due to implementation constraints in FLTK 1.3.x and maybe later. The reason is the way Fl::scheme() initializes the window's label type and image. A possible workaround is to use another Fl_Group as the only child widget and to set the background image for this group as described above.

Todo Fix Fl_Tiled_Image as background image for widgets and windows and fix the implementation of Fl::scheme(const char ∗).

31.139.3 Member Function Documentation

void Fl_Tiled_Image::color_average (Fl_Color *c*, float *i*) `[virtual]`

The color_average() method averages the colors in the image with the FLTK color value c.

The i argument specifies the amount of the original image to combine with the color, so a value of 1.0 results in no color blend, and a value of 0.0 results in a constant image of the specified color.

An internal copy is made of the original image before changes are applied, to avoid modifying the original image.

Reimplemented from Fl_Image.

Fl_Image ∗ Fl_Tiled_Image::copy (int *W*, int *H*) `[virtual]`

The copy() method creates a copy of the specified image.

If the width and height are provided, the image is resized to the specified size. The image should be deleted (or in the case of Fl_Shared_Image, released) when you are done with it.

Reimplemented from Fl_Image.

void Fl_Tiled_Image::desaturate ( ) `[virtual]`

The desaturate() method converts an image to grayscale.

If the image contains an alpha channel (depth = 4), the alpha channel is preserved.

An internal copy is made of the original image before changes are applied, to avoid modifying the original image.

Reimplemented from Fl_Image.

void Fl_Tiled_Image::draw (int *X*, int *Y*, int *W*, int *H*, int *cx*, int *cy*) `[virtual]`

Draws a tiled image.

Tiled images can be used as background images for widgets and windows. However, due to implementation constraints, you must take care when setting label types and alignment flags. Only certain combinations work as expected, others may yield unexpected results and undefined behavior.

This draw method can draw multiple copies of one image in an area given by `X`, `Y`, `W`, `H`.

The optional arguments `cx` and `cy` can be used to crop the image starting at offsets (cx, cy). `cx` and `cy` must be >= 0 (negative values are ignored). If one of the values is greater than the image width or height resp. (`cx` >= image()->w() or `cy` >= image()->h()) nothing is drawn, because the resulting image would be empty.

After calculating the resulting image size the image is drawn as often as necessary to fill the given area, starting at the top left corner.

If both `W` and `H` are 0 the image is repeated as often as necessary to fill the entire window, unless there is a valid clip region. If you want to fill only one particular widget's background, then you should either set a clip region in your draw() method or use the label alignment flags `FL_ALIGN_INSIDE|FL_ALIGN_CLIP` to make sure the image is clipped.

This may be improved in a later version of the library.

Reimplemented from Fl_Image.

The documentation for this class was generated from the following files:

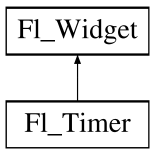
- Fl_Tiled_Image.H
- Fl_Tiled_Image.cxx

31.140 Fl_Timer Class Reference

This is provided only to emulate the Forms Timer widget.

`#include <Fl_Timer.H>`

Inheritance diagram for Fl_Timer:



Public Member Functions

- char direction () const

  Gets or sets the direction of the timer.
- void direction (char d)

  Gets or sets the direction of the timer.
- Fl_Timer (uchar t, int x, int y, int w, int h, const char ∗l)

  Creates a new Fl_Timer widget using the given type, position, size, and label string.
- int handle (int)

  Handles the specified event.
- char suspended () const

  Gets or sets whether the timer is suspended.
- void suspended (char d)

  Gets or sets whether the timer is suspended.
- void value (double)

Sets the current timer value.

- double value () const

  See void Fl_Timer::value(double)

- ∼Fl_Timer ()

  Destroys the timer and removes the timeout.

Protected Member Functions

- void draw ()

  Draws the widget.

Additional Inherited Members

31.140.1 Detailed Description

This is provided only to emulate the Forms Timer widget.

It works by making a timeout callback every 1/5 second. This is wasteful and inaccurate if you just want something to happen a fixed time in the future. You should directly call Fl::add_timeout() instead.

31.140.2 Constructor & Destructor Documentation

Fl_Timer::Fl_Timer (uchar *t*, int *X*, int *Y*, int *W*, int *H*, const char ∗ *l*)

Creates a new Fl_Timer widget using the given type, position, size, and label string.

The type parameter can be any of the following symbolic constants:

- FL_NORMAL_TIMER - The timer just does the callback and displays the string "Timer" in the widget.
- FL_VALUE_TIMER - The timer does the callback and displays the current timer value in the widget.
- FL_HIDDEN_TIMER - The timer just does the callback and does not display anything.

31.140.3 Member Function Documentation

char Fl_Timer::direction ( ) const `[inline]`

Gets or sets the direction of the timer.

If the direction is zero then the timer will count up, otherwise it will count down from the initial value().

void Fl_Timer::direction (char *d*) `[inline]`

Gets or sets the direction of the timer.

If the direction is zero then the timer will count up, otherwise it will count down from the initial value().

void Fl_Timer::draw ( ) `[protected], [virtual]`

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call redraw() instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw() method*, e.g. for an embedded scrollbar, you can do it (because draw() is virtual) like this:

```
Fl_Widget *s = &scroll;              // scroll is an embedded Fl_Scrollbar
s->draw();                  // calls Fl_Scrollbar::draw()
```

Implements Fl_Widget.

int Fl_Timer::handle (int *event*) [virtual]

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited handle() method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters

in	*event*	the kind of event received

Return values

0	if the event was not used or understood
1	if the event was used and can be deleted

See Also

Fl_Event

Reimplemented from Fl_Widget.

char Fl_Timer::suspended ( ) const [inline]

Gets or sets whether the timer is suspended.

void Fl_Timer::suspended (char *d*)

Gets or sets whether the timer is suspended.

The documentation for this class was generated from the following files:

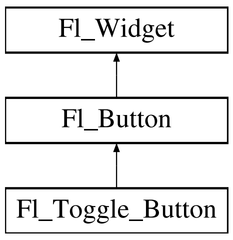
- Fl_Timer.H
- forms_timer.cxx

31.141 Fl_Toggle_Button Class Reference

The toggle button is a push button that needs to be clicked once to toggle on, and one more time to toggle off.

`#include <Fl_Toggle_Button.H>`

Inheritance diagram for Fl_Toggle_Button:



Public Member Functions

- Fl_Toggle_Button (int X, int Y, int W, int H, const char ∗l=0)

  Creates a new Fl_Toggle_Button widget using the given position, size, and label string.

Additional Inherited Members

31.141.1 Detailed Description

The toggle button is a push button that needs to be clicked once to toggle on, and one more time to toggle off.

The Fl_Toggle_Button subclass displays the "on" state by drawing a pushed-in button.

Buttons generate callbacks when they are clicked by the user. You control exactly when and how by changing the values for type() and when().

31.141.2 Constructor & Destructor Documentation

Fl_Toggle_Button::Fl_Toggle_Button (int *X*, int *Y*, int *W*, int *H*, const char ∗ *L* = `0`)

Creates a new Fl_Toggle_Button widget using the given position, size, and label string.

The constructor creates the button using the given position, size, and label.

The inherited destructor deletes the toggle button.

The Button type() is set to FL_TOGGLE_BUTTON.

Parameters

`in`	*X,Y,W,H*	position and size of the widget
`in`	*L*	widget label, default is no label

The documentation for this class was generated from the following files:

- Fl_Toggle_Button.H
- Fl_Button.cxx

31.142 Fl_Tooltip Class Reference

The Fl_Tooltip class provides tooltip support for all FLTK widgets.

`#include <Fl_Tooltip.H>`

Static Public Member Functions

- static Fl_Color color ()

  Gets the background color for tooltips.
- static void color (Fl_Color c)

  Sets the background color for tooltips.
- static Fl_Widget ∗ current ()

  Gets the current widget target.
- static void current (Fl_Widget ∗)

  Sets the current widget target.
- static float delay ()

  Gets the tooltip delay.
- static void delay (float f)

  Sets the tooltip delay.
- static void disable ()

Same as enable(0), disables tooltips on all widgets.

- static void enable (int b=1)

  Enables tooltips on all widgets (or disables if b is false).
- static int enabled ()

  Returns non-zero if tooltips are enabled.
- static void enter_area (Fl_Widget ∗w, int X, int Y, int W, int H, const char ∗tip)

  You may be able to use this to provide tooltips for internal pieces of your widget.
- static Fl_Font font ()

  Gets the typeface for the tooltip text.
- static void font (Fl_Font i)

  Sets the typeface for the tooltip text.
- static float hoverdelay ()

  Gets the tooltip hover delay, the delay between tooltips.
- static void hoverdelay (float f)

  Sets the tooltip hover delay, the delay between tooltips.
- static int margin_height ()

  Gets the amount of extra space above and below the tooltip's text.
- static void margin_height (int v)

  Sets the amount of extra space above and below the tooltip's text.
- static int margin_width ()

  Gets the amount of extra space left/right of the tooltip's text.
- static void margin_width (int v)

  Sets the amount of extra space left/right of the tooltip's text.
- static Fl_Fontsize size ()

  Gets the size of the tooltip text.
- static void size (Fl_Fontsize s)

  Sets the size of the tooltip text.
- static Fl_Color textcolor ()

  Gets the color of the text in the tooltip.
- static void textcolor (Fl_Color c)

  Sets the color of the text in the tooltip.
- static int wrap_width ()

  Gets the maximum width for tooltip's text before it word wraps.
- static void wrap_width (int v)

  Sets the maximum width for tooltip's text before it word wraps.

Static Public Attributes

- static void(∗ **enter**)(Fl_Widget ∗w) = nothing
- static void(∗ **exit**)(Fl_Widget ∗w) = nothing

Friends

- void **Fl_Widget::copy_tooltip** (const char ∗)
- void **Fl_Widget::tooltip** (const char ∗)

31.142.1 Detailed Description

The Fl_Tooltip class provides tooltip support for all FLTK widgets.

It contains only static methods.

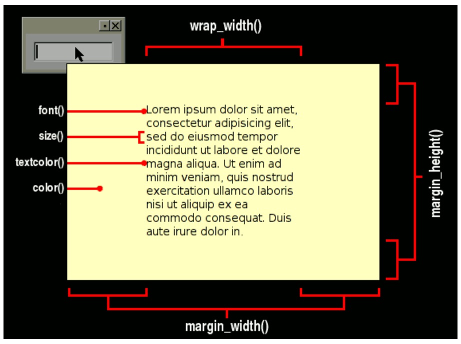

wrap_width()
font()
size()
textcolor()
color()
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in.
margin_height()
margin_width()

Figure 31.44: Fl_Tooltip Options

31.142.2 Member Function Documentation

static Fl_Color Fl_Tooltip::color ( ) `[inline], [static]`

Gets the background color for tooltips.

The default background color is a pale yellow.

static void Fl_Tooltip::color (Fl_Color *c*) `[inline], [static]`

Sets the background color for tooltips.

The default background color is a pale yellow.

void Fl_Tooltip::current (Fl_Widget ∗ *w*) `[static]`

Sets the current widget target.

Acts as though enter(widget) was done but does not pop up a tooltip. This is useful to prevent a tooltip from reappearing when a modal overlapping window is deleted. FLTK does this automatically when you click the mouse button.

static float Fl_Tooltip::delay ( ) `[inline], [static]`

Gets the tooltip delay.

The default delay is 1.0 seconds.

static void Fl_Tooltip::delay (float *f*) `[inline], [static]`

Sets the tooltip delay.

The default delay is 1.0 seconds.

static void Fl_Tooltip::disable ( ) `[inline], [static]`

Same as enable(0), disables tooltips on all widgets.

static void Fl_Tooltip::enable (int *b* = *1*) `[inline], [static]`

Enables tooltips on all widgets (or disables if *b* is false).

static int Fl_Tooltip::enabled ( ) `[inline]`, `[static]`

Returns non-zero if tooltips are enabled.

void Fl_Tooltip::enter_area (Fl_Widget ∗ *wid*, int *x*, int *y*, int *w*, int *h*, const char ∗ *t*) `[static]`

You may be able to use this to provide tooltips for internal pieces of your widget.

Call this after setting Fl::belowmouse() to your widget (because that calls the above enter() method). Then figure out what thing the mouse is pointing at, and call this with the widget (this pointer is used to remove the tooltip if the widget is deleted or hidden, and to locate the tooltip), the rectangle surrounding the area, relative to the top-left corner of the widget (used to calculate where to put the tooltip), and the text of the tooltip (which must be a pointer to static data as it is not copied).

static Fl_Font Fl_Tooltip::font ( ) `[inline]`, `[static]`

Gets the typeface for the tooltip text.

static void Fl_Tooltip::font (Fl_Font *i*) `[inline]`, `[static]`

Sets the typeface for the tooltip text.

static float Fl_Tooltip::hoverdelay ( ) `[inline]`, `[static]`

Gets the tooltip hover delay, the delay between tooltips.

The default delay is 0.2 seconds.

static void Fl_Tooltip::hoverdelay (float *f*) `[inline]`, `[static]`

Sets the tooltip hover delay, the delay between tooltips.

The default delay is 0.2 seconds.

static int Fl_Tooltip::margin_height ( ) `[inline]`, `[static]`

Gets the amount of extra space above and below the tooltip's text.

Default is 3.

static void Fl_Tooltip::margin_height (int *v*) `[inline]`, `[static]`

Sets the amount of extra space above and below the tooltip's text.

Default is 3.

static int Fl_Tooltip::margin_width ( ) `[inline]`, `[static]`

Gets the amount of extra space left/right of the tooltip's text.

Default is 3.

static void Fl_Tooltip::margin_width (int *v*) `[inline]`, `[static]`

Sets the amount of extra space left/right of the tooltip's text.

Default is 3.

static Fl_Fontsize Fl_Tooltip::size ( ) `[inline]`, `[static]`

Gets the size of the tooltip text.

static void Fl_Tooltip::size (Fl_Fontsize *s*) `[inline]`, `[static]`

Sets the size of the tooltip text.

static Fl_Color Fl_Tooltip::textcolor ( ) `[inline]`, `[static]`

Gets the color of the text in the tooltip.
The default is black.

static void Fl_Tooltip::textcolor (Fl_Color *c*) `[inline]`, `[static]`

Sets the color of the text in the tooltip.
The default is black.

static int Fl_Tooltip::wrap_width ( ) `[inline]`, `[static]`

Gets the maximum width for tooltip's text before it word wraps.
Default is 400.

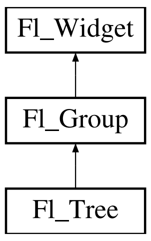
static void Fl_Tooltip::wrap_width (int *v*) `[inline]`, `[static]`

Sets the maximum width for tooltip's text before it word wraps.
Default is 400.
The documentation for this class was generated from the following files:

- Fl_Tooltip.H
- Fl.cxx
- Fl_Tooltip.cxx

31.143 Fl_Tree Class Reference

Tree widget.
`#include <Fl_Tree.H>`
Inheritance diagram for Fl_Tree:



Public Member Functions

- Fl_Tree_Item ∗ add (const char ∗path, Fl_Tree_Item ∗newitem=0)

  Adds a new item, given a menu style `'path'`*.*
- Fl_Tree_Item ∗ add (Fl_Tree_Item ∗parent_item, const char ∗name)

  Add a new child item labeled `'name'` *to the specified* `'parent_item'`*.*
- void calc_dimensions ()

  Recalculate widget dimensions and scrollbar visibility, normally managed automatically.
- void calc_tree ()

Recalculates the tree's sizes and scrollbar visibility, normally managed automatically.

- void callback_item (Fl_Tree_Item ∗item)

  Sets the item that was changed for this callback.
- Fl_Tree_Item ∗ callback_item ()

  Gets the item that caused the callback.
- void callback_reason (Fl_Tree_Reason reason)

  Sets the reason for this callback.
- Fl_Tree_Reason callback_reason () const

  Gets the reason for this callback.
- void clear ()

  Clear the entire tree's children, including the root.
- void clear_children (Fl_Tree_Item ∗item)

  Clear all the children for `'item'`.
- int close (Fl_Tree_Item ∗item, int docallback=1)

  Closes the specified `'item'`.
- int close (const char ∗path, int docallback=1)

  Closes the item specified by `'path'`.
- Fl_Image ∗ closeicon () const

  Returns the icon to be used as the 'close' icon.
- void closeicon (Fl_Image ∗val)

  Sets the icon to be used as the 'close' icon.
- Fl_Color connectorcolor () const

  Get the connector color used for tree connection lines.
- void connectorcolor (Fl_Color val)

  Set the connector color used for tree connection lines.
- Fl_Tree_Connector connectorstyle () const

  Returns the line drawing style for inter-connecting items.
- void connectorstyle (Fl_Tree_Connector val)

  Sets the line drawing style for inter-connecting items.
- int connectorwidth () const

  Gets the width of the horizontal connection lines (in pixels) that appear to the left of each tree item's label.
- void connectorwidth (int val)

  Sets the width of the horizontal connection lines (in pixels) that appear to the left of each tree item's label.
- int deselect (Fl_Tree_Item ∗item, int docallback=1)

  Deselect the specified `item`.
- int deselect (const char ∗path, int docallback=1)

  Deselect an item specified by `'path'`.
- int deselect_all (Fl_Tree_Item ∗item=0, int docallback=1)

  Deselect `'item'` *and all its children.*
- void display (Fl_Tree_Item ∗item)

  Displays `'item'`, *scrolling the tree as necessary.*
- int displayed (Fl_Tree_Item ∗item)

  See if `'item'` *is currently displayed on-screen (visible within the widget).*
- void draw ()

  Standard FLTK draw() method, handles drawing the tree widget.
- int extend_selection (Fl_Tree_Item ∗from, Fl_Tree_Item ∗to, int val=1, bool visible=false)

Extend a selection between `'from'` *and* `'to'` *depending on* `'visible'`*.*

- int extend_selection_dir (Fl_Tree_Item ∗from, Fl_Tree_Item ∗to, int dir, int val, bool visible)

  Extend the selection between and including `'from'` *and* `'to'` *depending on direction* `'dir'`*,* `'val'`*, and* `'visible'`*.*
- const Fl_Tree_Item ∗ find_clicked (int yonly=0) const

  Find the item that was last clicked on.
- Fl_Tree_Item ∗ find_clicked (int yonly=0)

  Non-const version of Fl_Tree::find_clicked(int yonly) const.
- Fl_Tree_Item ∗ find_item (const char ∗path)

  Non-const version of Fl_Tree::find_item(const char ∗path) const.
- const Fl_Tree_Item ∗ find_item (const char ∗path) const

  Find the item, given a menu style path, e.g.
- Fl_Tree_Item ∗ first ()

  Returns the first item in the tree, or 0 if none.
- Fl_Tree_Item ∗ first_selected_item ()

  Returns the first selected item in the tree.
- Fl_Tree_Item ∗ first_visible ()

  Returns the first open(), visible item in the tree, or 0 if none.
- Fl_Tree_Item ∗ first_visible_item ()

  Returns the first open(), visible item in the tree, or 0 if none.
- Fl_Tree (int X, int Y, int W, int H, const char ∗L=0)

  Constructor.
- Fl_Tree_Item ∗ get_item_focus () const

  Get the item that currently has keyboard focus.
- int get_selected_items (Fl_Tree_Item_Array &ret_items)

  Returns the currently selected items as an array of `'ret_items'`*.*
- int handle (int e)

  Standard FLTK event handler for this widget.
- int hposition () const

  Returns the horizontal scroll position as a pixel offset.
- void hposition (int pos)

  Sets the horizontal scroll offset to position `'pos'`*.*
- Fl_Tree_Item ∗ insert (Fl_Tree_Item ∗item, const char ∗name, int pos)

  Insert a new item `'name'` *into* `'item'`*s children at position* `'pos'`*.*
- Fl_Tree_Item ∗ insert_above (Fl_Tree_Item ∗above, const char ∗name)

  Inserts a new item `'name'` *above the specified Fl_Tree_Item* `'above'`*.*
- int is_close (Fl_Tree_Item ∗item) const

  See if the specified `'item'` *is closed.*
- int is_close (const char ∗path) const

  See if item specified by `'path'` *is closed.*
- int is_hscroll_visible () const

  See if the horizontal scrollbar is currently visible.
- int is_open (Fl_Tree_Item ∗item) const

  See if `'item'` *is open.*
- int is_open (const char ∗path) const

  See if item specified by `'path'` *is open.*
- int is_scrollbar (Fl_Widget ∗w)

See if widget `'w'` *is one of the Fl_Tree widget's scrollbars.*

- int is_selected (Fl_Tree_Item ∗item) const

  See if the specified `'item'` *is selected.*
- int is_selected (const char ∗path)

  See if item specified by `'path'` *is selected.*
- int is_vscroll_visible () const

  See if the vertical scrollbar is currently visible.
- Fl_Tree_Item ∗ item_clicked ()

  Return the item that was last clicked.
- Fl_Tree_Item_Draw_Mode item_draw_mode () const

  Get the 'item draw mode' used for the tree.
- void item_draw_mode (Fl_Tree_Item_Draw_Mode mode)

  Set the 'item draw mode' used for the tree to `'mode'`*.*
- void item_draw_mode (int mode)

  Set the 'item draw mode' used for the tree to integer `'mode'`*.*
- Fl_Color item_labelbgcolor (void) const

  Get the default label background color used for creating new items.
- void item_labelbgcolor (Fl_Color val)

  Set the default label background color used for creating new items.
- Fl_Color item_labelfgcolor (void) const

  Get the default label foreground color used for creating new items.
- void item_labelfgcolor (Fl_Color val)

  Set the default label foreground color used for creating new items.
- Fl_Font item_labelfont () const

  Get the default font face used for creating new items.
- void item_labelfont (Fl_Font val)

  Set the default font face used for creating new items.
- Fl_Fontsize item_labelsize () const

  Get the default label fontsize used for creating new items.
- void item_labelsize (Fl_Fontsize val)

  Set the default label font size used for creating new items.
- int item_pathname (char ∗pathname, int pathnamelen, const Fl_Tree_Item ∗item) const

  Return `'pathname'` *of size* `'pathnamelen'` *for the specified* `'item'`*.*
- Fl_Tree_Item_Reselect_Mode item_reselect_mode () const

  Returns the current item re/selection mode.
- void item_reselect_mode (Fl_Tree_Item_Reselect_Mode mode)

  Sets the item re/selection mode.
- int labelmarginleft () const

  Get the amount of white space (in pixels) that should appear to the left of the label text.
- void labelmarginleft (int val)

  Set the amount of white space (in pixels) that should appear to the left of the label text.
- Fl_Tree_Item ∗ last ()

  Returns the last item in the tree.
- Fl_Tree_Item ∗ last_selected_item ()

  Returns the last selected item in the tree.
- Fl_Tree_Item ∗ last_visible ()

Returns the last open(), visible item in the tree.

- Fl_Tree_Item ∗ last_visible_item ()

  Returns the last open(), visible item in the tree.

- int linespacing () const

  Get the amount of white space (in pixels) that should appear between items in the tree.

- void linespacing (int val)

  Sets the amount of white space (in pixels) that should appear between items in the tree.

- void load (class Fl_Preferences &)

  Load FLTK preferences.

- int marginbottom () const

  Get the amount of white space (in pixels) that should appear below the last visible item when the vertical scroller is scrolled to the bottom.

- void marginbottom (int val)

  Sets the amount of white space (in pixels) that should appear below the last visible item when the vertical scroller is scrolled to the bottom.

- int marginleft () const

  Get the amount of white space (in pixels) that should appear between the widget's left border and the tree's contents.

- void marginleft (int val)

  Set the amount of white space (in pixels) that should appear between the widget's left border and the left side of the tree's contents.

- int margintop () const

  Get the amount of white space (in pixels) that should appear between the widget's top border and the top of the tree's contents.

- void margintop (int val)

  Sets the amount of white space (in pixels) that should appear between the widget's top border and the top of the tree's contents.

- Fl_Tree_Item ∗ next (Fl_Tree_Item ∗item=0)

  Return the next item after `'item'`, or 0 if no more items.

- Fl_Tree_Item ∗ next_item (Fl_Tree_Item ∗item, int dir=FL_Down, bool visible=false)

  Returns next item after `'item'` in direction `'dir'` depending on `'visible'`.

- Fl_Tree_Item ∗ next_selected_item (Fl_Tree_Item ∗item=0, int dir=FL_Down)

  Returns the next selected item above or below `'item'`, depending on `'dir'`.

- Fl_Tree_Item ∗ next_visible_item (Fl_Tree_Item ∗start, int dir)

  Returns next open(), visible item above (`dir==FL_Up`) or below (`dir==FL_Down`) the specified `'item'`, or 0 if no more items.

- int open (Fl_Tree_Item ∗item, int docallback=1)

  Open the specified `'item'`.

- int open (const char ∗path, int docallback=1)

  Opens the item specified by `'path'`.

- void open_toggle (Fl_Tree_Item ∗item, int docallback=1)

  Toggle the open state of `'item'`.

- int openchild_marginbottom () const

  Get the amount of white space (in pixels) that should appear below an open child tree's contents.

- void openchild_marginbottom (int val)

  Set the amount of white space (in pixels) that should appear below an open child tree's contents.

- Fl_Image ∗ openicon () const

  Returns the icon to be used as the 'open' icon.

- void openicon (Fl_Image ∗val)

  Sets the icon to be used as the 'open' icon.
- const Fl_Tree_Prefs & **prefs** () const
- Fl_Tree_Item ∗ prev (Fl_Tree_Item ∗item=0)

  Return the previous item before `'item'`, or 0 if no more items.
- void recalc_tree ()

  Schedule tree to recalc the entire tree size.
- int remove (Fl_Tree_Item ∗item)

  Remove the specified `'item'` from the tree.
- void resize (int, int, int, int)

  Resizes the Fl_Group widget and all of its children.
- Fl_Tree_Item ∗ root ()

  Returns the root item.
- void root (Fl_Tree_Item ∗newitem)

  Sets the root item to `'newitem'`.
- void root_label (const char ∗new_label)

  Set the label for the root item to `'new_label'`.
- int scrollbar_size () const

  Gets the default size of scrollbars' troughs for this widget in pixels.
- void scrollbar_size (int size)

  Sets the pixel size of the scrollbars' troughs to `'size'` for this widget, in pixels.
- int select (Fl_Tree_Item ∗item, int docallback=1)

  Select the specified `'item'`.
- int select (const char ∗path, int docallback=1)

  Select the item specified by `'path'`.
- int select_all (Fl_Tree_Item ∗item=0, int docallback=1)

  Select `'item'` and all its children.
- int select_only (Fl_Tree_Item ∗selitem, int docallback=1)

  Select only the specified item, deselecting all others that might be selected.
- void select_toggle (Fl_Tree_Item ∗item, int docallback=1)

  Toggle the select state of the specified `'item'`.
- Fl_Boxtype selectbox () const

  Sets the style of box used to draw selected items.
- void selectbox (Fl_Boxtype val)

  Gets the style of box used to draw selected items.
- Fl_Tree_Select selectmode () const

  Gets the tree's current selection mode.
- void selectmode (Fl_Tree_Select val)

  Sets the tree's selection mode.
- void set_item_focus (Fl_Tree_Item ∗item)

  Set the item that currently should have keyboard focus.
- void show_item (Fl_Tree_Item ∗item, int yoff)

  Adjust the vertical scrollbar so that `'item'` is visible `'yoff'` pixels from the top of the Fl_Tree widget's display.
- void show_item (Fl_Tree_Item ∗item)

  Adjust the vertical scrollbar to show `'item'` at the top of the display IF it is currently off-screen (for instance show_item_top()).

- void show_item_bottom (Fl_Tree_Item ∗item)

  Adjust the vertical scrollbar so that `'item'` *is at the bottom of the display.*
- void show_item_middle (Fl_Tree_Item ∗item)

  Adjust the vertical scrollbar so that `'item'` *is in the middle of the display.*
- void show_item_top (Fl_Tree_Item ∗item)

  Adjust the vertical scrollbar so that `'item'` *is at the top of the display.*
- void show_self ()

  Print the tree as 'ascii art' to stdout.
- int showcollapse () const

  Returns 1 if the collapse icon is enabled, 0 if not.
- void showcollapse (int val)

  Set if we should show the collapse icon or not.
- int showroot () const

  Returns 1 if the root item is to be shown, or 0 if not.
- void showroot (int val)

  Set if the root item should be shown or not.
- Fl_Tree_Sort sortorder () const

  Set the default sort order used when items are added to the tree.
- void sortorder (Fl_Tree_Sort val)

  Gets the sort order used to add items to the tree.
- Fl_Image ∗ usericon () const

  Returns the Fl_Image being used as the default user icon for all newly created items.
- void usericon (Fl_Image ∗val)

  Sets the Fl_Image to be used as the default user icon for all newly created items.
- int usericonmarginleft () const

  Get the amount of white space (in pixels) that should appear to the left of the usericon.
- void usericonmarginleft (int val)

  Set the amount of white space (in pixels) that should appear to the left of the usericon.
- int vposition () const

  Returns the vertical scroll position as a pixel offset.
- void vposition (int pos)

  Sets the vertical scroll offset to position `'pos'`*.*
- int widgetmarginleft () const

  Get the amount of white space (in pixels) that should appear to the left of the child fltk widget (if any).
- void widgetmarginleft (int val)

  Set the amount of white space (in pixels) that should appear to the left of the child fltk widget (if any).
- ∼Fl_Tree ()

  Destructor.

Protected Member Functions

- void do_callback_for_item (Fl_Tree_Item ∗item, Fl_Tree_Reason reason)

  Do the callback for the specified `'item'` *using* `'reason'`*, setting the callback_item() and callback_reason().*
- void item_clicked (Fl_Tree_Item ∗val)

  Set the item that was last clicked.

Protected Attributes

- Fl_Scrollbar ∗ _hscroll

  Horizontal scrollbar.
- int _tih

  Tree widget inner xywh dimension: inside borders + scrollbars.
- int **_tiw**
- int **_tix**
- int **_tiy**
- int _toh

  Tree widget outer xywh dimension: outside scrollbars, inside widget border.
- int **_tow**
- int **_tox**
- int **_toy**
- int _tree_h

  the calculated height of the entire tree hierarchy. See calc_tree()
- int _tree_w

  the calculated width of the entire tree hierarchy. See calc_tree()
- Fl_Scrollbar ∗ _vscroll

  Vertical scrollbar.

Friends

- class **Fl_Tree_Item**

Additional Inherited Members

31.143.1 Detailed Description

Tree widget.

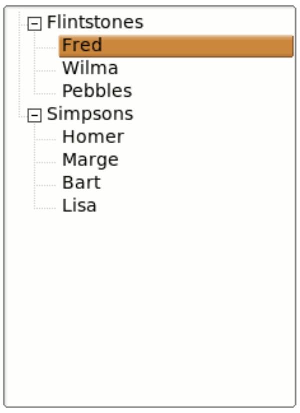


Figure 31.45: Fl_Tree example program

```
Fl_Tree                                         // Top level widget
   |--- Fl_Tree_Item                           // Items in the tree
   |--- Fl_Tree_Prefs                          // Preferences for the tree
             |--- Fl_Tree_Connector (enum)     // Connection modes
             |--- Fl_Tree_Select (enum)        // Selection modes
             |--- Fl_Tree_Sort (enum)          // Sort behavior
```

Similar to Fl_Browser, Fl_Tree is a browser of Fl_Tree_Item's arranged in a parented hierarchy, or 'tree'. Subtrees can be expanded or closed. Items can be added, deleted, inserted, sorted and re-ordered.

The tree items may also contain other FLTK widgets, like buttons, input fields, or even "custom" widgets.

The callback() is invoked depending on the value of when():

```
- FL_WHEN_RELEASE -- callback invoked when left mouse button is released on an item
- FL_WHEN_CHANGED -- callback invoked when left mouse changes selection state
```

The simple way to define a tree:

```
#include <FL/Fl_Tree.H>
[..]
Fl_Tree tree(X,Y,W,H);
tree.begin();
  tree.add("Flintstones/Fred");
  tree.add("Flintstones/Wilma");
  tree.add("Flintstones/Pebbles");
  tree.add("Simpsons/Homer");
  tree.add("Simpsons/Marge");
  tree.add("Simpsons/Bart");
  tree.add("Simpsons/Lisa");
tree.end();
```

FEATURES

Items can be added with add(),
removed with remove(),
completely cleared with clear(),
inserted with insert() and insert_above(),
selected/deselected with select() and deselect(),
open/closed with open() and close(),
positioned on the screen with show_item_top(), show_item_middle() and show_item_bottom(),
item children can be swapped around with Fl_Tree_Item::swap_children(),
sorting can be controlled when items are add()ed via sortorder().
You can walk the entire tree with first() and next().
You can walk visible items with first_visible_item() and next_visible_item().
You can walk selected items with first_selected_item() and next_selected_item().
Items can be found by their pathname using find_item(const char*), and an item's pathname can be found with item_pathname().
The selected items' colors are controlled by selection_color() (inherited from Fl_Widget).
A hook is provided to allow you to redefine how item's labels are drawn via Fl_Tree::item_draw_-callback().

SELECTION OF ITEMS

The tree can have different selection behaviors controlled by selectmode(). The background color used for selected items is the Fl_Tree::selection_color(). The foreground color for selected items is controlled internally with fl_contrast().

CHILD WIDGETS

FLTK widgets (including custom widgets) can be assigned to tree items via Fl_Tree_Item::widget().

When an Fl_Tree_Item::widget() is defined, the default behavior is for the widget() to be shown in place of the item's label (if it has one). Only the widget()'s width will be used; the widget()'s x() and y() position will be managed by the tree, and the h() will track the item's height. This default behavior can be altered (ABI 1.3.1): Setting Fl_Tree::item_draw_mode()'s FL_TREE_ITEM_DRAW_LABEL_AND_WIDGET flag causes the label + widget to be displayed together in that order, and adding the FL_TREE_ITEM_HEIGHT_FROM_WIDGET flag causes widget's height to define the widget()'s height.

ICONS

The tree's open/close icons can be redefined with Fl_Tree::openicon(), Fl_Tree::closeicon(). User icons can either be changed globally with Fl_Tree::usericon(), or on a per-item basis with Fl_Tree_Item::usericon().

Various default preferences can be globally manipulated via Fl_Tree_Prefs, including colors, margins, icons, connection lines, etc.

FONTS AND COLORS

When adding new items to the tree, the new items get the defaults for fonts and colors from:

- Fl_Tree::item_labelfont() – The default item label font (default: FL_HELVETICA)
- Fl_Tree::item_labelsize() – The default item label size (default: FL_NORMAL_SIZE)
- Fl_Tree::item_labelfgcolor() – The default item label foreground color (default: FL_FOREGROUND_COLOR)
- Fl_Tree::item_labelbgcolor() – The default item label background color (default: 0xffffffff, which tree uses as 'transparent')

Each item (Fl_Tree_Item) inherits a copy of these font/color attributes when created, and each item has its own methods to let the app change these values on a per-item basis using methods of the same name:

- Fl_Tree_Item::labelfont() – The item's label font (default: FL_HELVETICA)
- Fl_Tree_Item::labelsize() – The item's label size (default: FL_NORMAL_SIZE)
- Fl_Tree_Item::labelfgcolor() – The item's label foreground color (default: FL_FOREGROUND_COLOR)
- Fl_Tree_Item::labelbgcolor() – The item's label background color (default: 0xffffffff, which uses the tree's own bg color)

CALLBACKS

The tree's callback() will be invoked when items change state or are open/closed. when() controls when mouse/keyboard events invoke the callback. callback_item() and callback_reason() can be used to determine the cause of the callback. e.g.

```
void MyTreeCallback(Fl_Widget *w, void *data) {
  Fl_Tree      *tree = (Fl_Tree*)w;
  Fl_Tree_Item *item = (Fl_Tree_Item*)tree->
      callback_item();    // get selected item
  switch ( tree->callback_reason() ) {
    case FL_TREE_REASON_SELECTED: [..]
    case FL_TREE_REASON_DESELECTED: [..]
    case FL_TREE_REASON_RESELECTED: [..]
    case FL_TREE_REASON_OPENED: [..]
    case FL_TREE_REASON_CLOSED: [..]
  }
```

SIMPLE EXAMPLES

To find all the selected items:

```
for ( Fl_Tree_Item *i=first_selected_item(); i; i=
      next_selected_item(i) )
  printf("Item %s is selected\n", i->label());
```

To get an item's full menu pathname, use Fl_Tree::item_pathname(), e.g.

```
char pathname[256] = "???";
tree->item_pathname(pathname, sizeof(pathname), item);                // eg.
       "Parent/Child/Item"
```

To walk all the items of the tree from top to bottom:

```
// Walk all the items in the tree, and print their labels
for ( Fl_Tree_Item *item = tree->first(); item; item = tree->
      next(item) ) {
    printf("Item: %s\n", item->label());
}
```

To recursively walk all the children of a particular item, define a function that uses recursion:

```
// Find all of the item's children and print an indented report of their labels
void my_print_all_children(Fl_Tree_Item *item, int indent=0) {
    for ( int t=0; t<item->children(); t++ ) {
        printf("%*s Item: %s\n", indent, "", item->child(t)->label());
        my_print_all_children(item->child(t), indent+4);   // recurse
    }
}
```

To change the default label font and color when creating new items:

```
tree = new Fl_Tree(..);
tree->item_labelfont(FL_COURIER);      // Use Courier font for all new items
tree->item_labelfgcolor(FL_RED);       // Use red color for labels of all new items
[..]
// Now create the items in the tree using the above defaults.
tree->add("Aaa");
tree->add("Bbb");
[..]
```

To change the font and color of all existing items in the tree:

```
// Change the font and color of all items currently in the tree
for ( Fl_Tree_Item *item = tree->first(); item; item = tree->
      next(item) ) {
    item->labelfont(FL_COURIER);
    item->labelcolor(FL_RED);
}
```

DISPLAY DESCRIPTION

The following image shows the tree's various visual elements and the methods that control them:

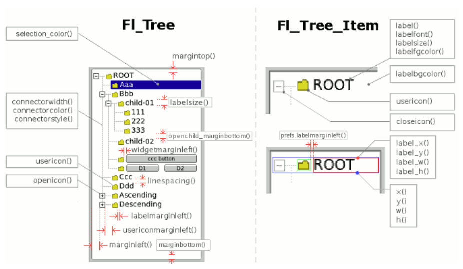


Figure 31.46: Fl_Tree elements

The following shows the protected dimension variables 'tree inner' (tix..) and 'tree outer' (tox..):

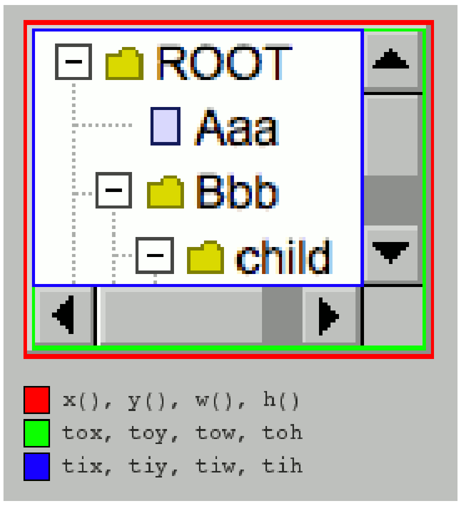


Figure 31.47: Fl_Tree inner/outer dimensions

KEYBOARD BINDINGS

The following table lists keyboard bindings for navigating the tree:

31.143.2 Member Function Documentation

Fl_Tree_Item ∗ Fl_Tree::add (const char ∗ *path*, Fl_Tree_Item ∗ *item* = 0)

Adds a new item, given a menu style `'path'`.

Any parent nodes that don't already exist are created automatically. Adds the item based on the value of sortorder(). If `'item'` is NULL, a new item is created.

To specify items or submenus that contain slashes ('/' or '\') use an escape character to protect them, e.g.

```
tree->add("/Holidays/Photos/12\\/25\\/2010");         // Adds item "12/25/2010"
tree->add("/Pathnames/c:\\\\Program Files\\\\MyApp"); // Adds item "c:\Program Files\MyApp"
```

Parameters

in	*path*	The path to the item, e.g. "Flintstone/Fred".
in	*item*	The new item to be added. If NULL, a new item is created with a name that is the last element in `'path'`.

Returns

The new item added, or 0 on error.

Version

1.3.3

Fl_Tree_Item ∗ Fl_Tree::add (Fl_Tree_Item ∗ *parent_item*, const char ∗ *name*)

Add a new child item labeled `'name'` to the specified `'parent_item'`.

Parameters

in	*parent_item*	The parent item the new child item will be added to. Must not be NULL.
in	*name*	The label for the new item

Returns

The new item added.

Version

1.3.0 release

void Fl_Tree::calc_dimensions ( )

Recalculate widget dimensions and scrollbar visibility, normally managed automatically.

Low overhead way to update the tree widget's outer/inner dimensions and re-determine scrollbar visibility based on these changes without recalculating the entire size of the tree data.

Assumes that either the tree's size in _tree_w/_tree_h are correct so that scrollbar visibility can be calculated easily, or are both zero indicating scrollbar visibility can't be calculated yet.

This method is called when the widget is resize()ed or if the scrollbar's sizes are changed (affects tree widget's inner dimensions tix/y/w/h), and also used by calc_tree().

Version

1.3.3 ABI feature

void Fl_Tree::calc_tree ( )

Recalculates the tree's sizes and scrollbar visibility, normally managed automatically.

On return:

```
- _tree_w will be the overall pixel width of the entire viewable tree
- _tree_h will be the overall pixel height ""
- scrollbar visibility and pan sizes are updated
- internal _tix/_tiy/_tiw/_tih dimensions are updated
```

_tree_w/_tree_h include the tree's margins (e.g. marginleft()), whether items are open or closed, label contents and font sizes, etc.

The tree hierarchy's size is managed separately from the widget's size as an optimization; this way resize() on the widget doesn't involve recalculating the tree's hierarchy needlessly, as widget size has no bearing on the tree hierarchy.

The tree hierarchy's size only changes when items are added/removed, open/closed, label contents or font sizes changed, margins changed, etc.

This calculation involves walking the *entire* tree from top to bottom, potentially a slow calculation if the tree has many items (potentially hundreds of thousands), and should therefore be called sparingly.

For this reason, recalc_tree() is used as a way to /schedule/ calculation when changes affect the tree hierarchy's size.

Apps may want to call this method directly if the app makes changes to the tree's geometry, then immediately needs to work with the tree's new dimensions before an actual redraw (and recalc) occurs. (This use by an app should only rarely be needed)

void Fl_Tree::callback_item (Fl_Tree_Item * *item*)

Sets the item that was changed for this callback.

Used internally to pass the item that invoked the callback.

Fl_Tree_Item ∗ Fl_Tree::callback_item ( )

Gets the item that caused the callback.
The callback() can use this value to see which item changed.

void Fl_Tree::callback_reason (Fl_Tree_Reason *reason*)

Sets the reason for this callback.
Used internally to pass the reason the callback was invoked.

Fl_Tree_Reason Fl_Tree::callback_reason ( ) const

Gets the reason for this callback.
The callback() can use this value to see why it was called. Example:

```
void MyTreeCallback(Fl_Widget *w, void *userdata) {
    Fl_Tree *tree = (Fl_Tree*)w;
    Fl_Tree_Item *item = tree->callback_item();    // the item changed (can be
       NULL if more than one item was changed!)
    switch ( tree->callback_reason() ) {           // reason callback was invoked
        case     FL_TREE_REASON_OPENED: ..item was opened..
        case     FL_TREE_REASON_CLOSED: ..item was closed..
        case   FL_TREE_REASON_SELECTED: ..item was selected..
        case FL_TREE_REASON_RESELECTED: ..item was reselected (double-clicked, etc
      )..
        case FL_TREE_REASON_DESELECTED: ..item was deselected..
    }
}
```

See Also

item_reselect_mode() – enables FL_TREE_REASON_RESELECTED events

void Fl_Tree::clear ( )

Clear the entire tree's children, including the root.
The tree will be left completely empty.

void Fl_Tree::clear_children (Fl_Tree_Item ∗ *item*)

Clear all the children for `'item'`.
Item may not be NULL.

int Fl_Tree::close (Fl_Tree_Item ∗ *item*, int *docallback* = 1)

Closes the specified `'item'`.
Invokes the callback depending on the value of optional parameter `'docallback'`.
Handles calling redraw() if anything changed.
The callback can use callback_item() and callback_reason() respectively to determine the item changed and the reason the callback was called.
Parameters

in	*item*	– the item to be closed. Must not be NULL.
in	*docallback*	– A flag that determines if the callback() is invoked or not: • 0 - callback() is not invoked • 1 - callback() is invoked if item changed (default), callback_reason() will be FL_TREE_REASON_CLOSED

Returns

- 1 – item was closed
- 0 – item was already closed, no change

See Also

open(), close(), is_open(), is_close(), callback_item(), callback_reason()

int Fl_Tree::close (const char ∗ *path*, int *docallback* = `1`)

Closes the item specified by `'path'`.

Invokes the callback depending on the value of optional parameter `'docallback'`.

Handles calling redraw() if anything changed.

Items or submenus that themselves contain slashes ('/' or '\') should be escaped, e.g. close("Holidays/12\\/25\\/2010").

The callback can use callback_item() and callback_reason() respectively to determine the item changed and the reason the callback was called.

Parameters

`in`	*path*	– the tree item's pathname (e.g. "Flintstones/Fred")
`in`	*docallback*	– A flag that determines if the callback() is invoked or not: • 0 - callback() is not invoked • 1 - callback() is invoked if item changed (default), callback_reason() will be FL_TREE_REASON_CLOSED

Returns

- 1 – OK: item closed
- 0 – OK: item was already closed, no change
- -1 – ERROR: item was not found

See Also

open(), close(), is_open(), is_close(), callback_item(), callback_reason()

Fl_Image ∗ Fl_Tree::closeicon ( ) const

Returns the icon to be used as the 'close' icon.

If none was set, the internal default is returned, a simple '[-]' icon.

void Fl_Tree::closeicon (Fl_Image ∗ *val*)

Sets the icon to be used as the 'close' icon.

This overrides the built in default '[-]' icon.

Parameters

`in`	*val*	– The new image, or zero to use the default [-] icon.

void Fl_Tree::connectorstyle (Fl_Tree_Connector *val*)

Sets the line drawing style for inter-connecting items.

See Fl_Tree_Connector for possible values.

int Fl_Tree::deselect (Fl_Tree_Item ∗ *item*, int *docallback* = 1)

Deselect the specified `item`.

Invokes the callback depending on the value of optional parameter `'docallback'`.

Handles calling redraw() if anything changed.

The callback can use callback_item() and callback_reason() respectively to determine the item changed and the reason the callback was called.

Parameters

`in`	*item*	– the item to be deselected. Must not be NULL.
`in`	*docallback*	– A flag that determines if the callback() is invoked or not: • 0 - the callback() is not invoked • 1 - the callback() is invoked if item changed state (default), callback_reason() will be FL_TREE_REASON_DESELECTED

Returns

- 0 - item was already deselected, no change was made
- 1 - item's state was changed

int Fl_Tree::deselect (const char ∗ *path*, int *docallback* = 1)

Deselect an item specified by `'path'`.

Invokes the callback depending on the value of optional parameter `'docallback'`.

Handles calling redraw() if anything changed.

Items or submenus that themselves contain slashes ('/' or '\') should be escaped, e.g. deselect("-Holidays/12\\/25\\/2010").

The callback can use callback_item() and callback_reason() respectively to determine the item changed and the reason the callback was called.

Parameters

`in`	*path*	– the tree item's pathname (e.g. "Flintstones/Fred")
`in`	*docallback*	– A flag that determines if the callback() is invoked or not: • 0 - the callback() is not invoked • 1 - the callback() is invoked if item changed state (default), callback_reason() will be FL_TREE_REASON_DESELECTED

Returns

- 1 - OK: item's state was changed
- 0 - OK: item was already deselected, no change was made
- -1 - ERROR: item was not found

int Fl_Tree::deselect_all (Fl_Tree_Item ∗ *item* = 0, int *docallback* = 1)

Deselect `'item'` and all its children.

If item is NULL, first() is used.

Invokes the callback depending on the value of optional parameter `'docallback'`.

Handles calling redraw() if anything changed.

The callback can use callback_item() and callback_reason() respectively to determine the item changed and the reason the callback was called.

Parameters

in	*item*	The item that will be deselected (along with all its children). If NULL, first() is used.
in	*docallback*	– A flag that determines if the callback() is invoked or not: • 0 - the callback() is not invoked • 1 - the callback() is invoked for each item that changed state (default), callback_reason() will be FL_TREE_REASON_DESELECTED

Returns

Count of how many items were actually changed to the deselected state.

void Fl_Tree::display (Fl_Tree_Item ∗ *item*)

Displays `'item'`, scrolling the tree as necessary.
Parameters

in	*item*	The item to be displayed. If NULL, first() is used.

int Fl_Tree::displayed (Fl_Tree_Item ∗ *item*)

See if `'item'` is currently displayed on-screen (visible within the widget).

This can be used to detect if the item is scrolled off-screen. Checks to see if the item's vertical position is within the top and bottom edges of the display window. This does NOT take into account the hide() / show() or open() / close() status of the item.
Parameters

in	*item*	The item to be checked. If NULL, first() is used.

Returns

1 if displayed, 0 if scrolled off screen or no items are in tree.

int Fl_Tree::extend_selection (Fl_Tree_Item ∗ *from*, Fl_Tree_Item ∗ *to*, int *val* = *1*, bool *visible* = *false*)

Extend a selection between `'from'` and `'to'` depending on `'visible'`.

Similar to the more efficient extend_selection_dir(Fl_Tree_Item∗,Fl_Tree_Item∗,int dir,int val,bool vis) method, but direction (up or down) doesn't need to be known.

We're less efficient because we search the tree for to/from, then operate on items in between. The more efficient method avoids the "search", but necessitates a direction to be specified to find `'to'`.

Used by SHIFT-click to extend a selection between two items inclusive.

Handles calling redraw() if anything changed.
Parameters

in	*from*	Starting item
in	*to*	Ending item
in	*val*	Select or deselect items (0=deselect, 1=select, 2=toggle)
in	*visible*	true=affect only open(), visible items, false=affect open or closed items (default)

Returns

The number of items whose selection states were changed, if any.

Version

1.3.3 ABI feature

int Fl_Tree::extend_selection_dir (Fl_Tree_Item ∗ *from*, Fl_Tree_Item ∗ *to*, int *dir*, int *val*, bool *visible*)

Extend the selection between and including `'from'` and `'to'` depending on direction `'dir'`, `'val'`, and `'visible'`.

Efficient: does not walk entire tree; starts with `'from'` and stops at `'to'` while moving in direction `'dir'`. Dir must be specified though. If dir cannot be known in advance, such as during SHIFT-click operations, the method extend_selection(Fl_Tree_Item∗,Fl_Tree_Item∗,int,bool) should be used. Handles calling redraw() if anything changed.

Parameters

`in`	*from*	Starting item
`in`	*to*	Ending item
`in`	*dir*	Direction to extend selection (FL_Up or FL_Down)
`in`	*val*	0=deselect, 1=select, 2=toggle
`in`	*visible*	true=affect only open(), visible items, false=affect open or closed items (default)

Returns

The number of items whose selection states were changed, if any.

Version

1.3.3

const Fl_Tree_Item ∗ Fl_Tree::find_clicked (int *yonly* = `0`) const

Find the item that was last clicked on.

You should use callback_item() instead, which is fast, and is meant to be used within a callback to determine the item clicked.

This method walks the entire tree looking for the first item that is under the mouse. (The value of the `'yonly'` flag affects whether both x and y events are checked, or just y)

Use this method /only/ if you've subclassed Fl_Tree, and are receiving events before Fl_Tree has been able to process and update callback_item().

Parameters

`in`	*yonly*	– 0: check both event's X and Y values. – 1: only check event's Y value, don't care about X.

Returns

The item clicked, or NULL if no item was under the current event.

Version

1.3.0
1.3.3 ABI feature: added yonly parameter

const Fl_Tree_Item ∗ Fl_Tree::find_item (const char ∗ *path*) const

Find the item, given a menu style path, e.g.

”/Parent/Child/item”. There is both a const and non-const version of this method. Const version allows pure const methods to use this method to do lookups without causing compiler errors.

To specify items or submenus that contain slashes (’/’ or ’\’) use an escape character to protect them, e.g.

```
tree->add("/Holidays/Photos/12\\/25\\/2010");          // Adds item "12/25/2010"
tree->add("/Pathnames/c:\\\\Program Files\\\\MyApp"); // Adds item "c:\Program Files\MyApp"
```

Parameters

in	*path*	– the tree item’s pathname to be found (e.g. ”Flintstones/Fred”)

Returns

The item, or NULL if not found.

See Also

item_pathname()

Fl_Tree_Item ∗ Fl_Tree::first ( )

Returns the first item in the tree, or 0 if none.

Use this to walk the tree in the forward direction, e.g.

```
for ( Fl_Tree_Item *item = tree->first(); item; item = tree->
      next(item) )
    printf("Item: %s\n", item->label());
```

Returns

First item in tree, or 0 if none (tree empty).

See Also

first(), next(), last(), prev()

Fl_Tree_Item ∗ Fl_Tree::first_selected_item ( )

Returns the first selected item in the tree.

Use this to walk the tree from top to bottom looking for all the selected items, e.g.

```
// Walk tree forward, from top to bottom
for ( Fl_Tree_Item *i=tree->first_selected_item(); i; i=tree->
      next_selected_item(i) )
    printf("Selected item: %s\n", i->label());
```

Returns

The first selected item, or 0 if none.

See Also

first_selected_item(), last_selected_item(), next_selected_item()

Fl_Tree_Item ∗ Fl_Tree::first_visible ( )

Returns the first open(), visible item in the tree, or 0 if none.

Deprecated in 1.3.3 ABI – use first_visible_item() instead.

Fl_Tree_Item ∗ Fl_Tree::first_visible_item ( )

Returns the first open(), visible item in the tree, or 0 if none.

Returns

First visible item in tree, or 0 if none.

See Also

first_visible_item(), last_visible_item(), next_visible_item()

Version

1.3.3

int Fl_Tree::get_selected_items (Fl_Tree_Item_Array & *ret_items*)

Returns the currently selected items as an array of `'ret_items'`.
Example:

```
// Get selected items as an array
Fl_Tree_Item_Array items;
tree->get_selected_items(items);
// Manipulate the returned array
for ( int t=0; t<items.total(); t++ ) {
    Fl_Tree_Item &item = items[t];
    ..do stuff with each selected item..
}
```

Parameters

out	*ret_items*	The returned array of selected items.

Returns

The number of items in the returned array.

See Also

first_selected_item(), next_selected_item()

Version

1.3.3 ABI feature

int Fl_Tree::handle (int *e*) [virtual]

Standard FLTK event handler for this widget.

Todo add Fl_Widget_Tracker (see Fl_Browser_.cxx::handle())

Reimplemented from Fl_Group.

int Fl_Tree::hposition ( ) const

Returns the horizontal scroll position as a pixel offset.

The position returned is how many pixels of the tree are scrolled off the left edge of the screen.

See Also

hposition(int), vposition(), vposition(int)

Note

Must be using FLTK ABI 1.3.3 or higher for this to be effective.

void Fl_Tree::hposition (int *pos*)

Sets the horizontal scroll offset to position `'pos'`.

The position is how many pixels of the tree are scrolled off the left edge of the screen.

Parameters

in	*pos*	The vertical position (in pixels) to scroll the tree to.

See Also

hposition(), vposition(), vposition(int)

Note

Must be using FLTK ABI 1.3.3 or higher for this to be effective.

Fl_Tree_Item ∗ Fl_Tree::insert (Fl_Tree_Item ∗ *item*, const char ∗ *name*, int *pos*)

Insert a new item `'name'` into `'item'`s children at position `'pos'`.

Example:

```
tree->add("Aaa/000");       // "000" is index 0 in Aaa's children
tree->add("Aaa/111");       // "111" is index 1 in Aaa's children
tree->add("Aaa/222");       // "222" is index 2 in Aaa's children
..
// How to use insert() to insert a new item between Aaa/111 + Aaa/222
Fl_Tree_Item *item = tree->find_item("Aaa");  // get parent item Aaa
if (item) tree->insert(item, "New item", 2);  // insert as a child of Aaa at index #2
```

Parameters

in	*item*	The existing item to insert new child into. Must not be NULL.
in	*name*	The label for the new item
in	*pos*	The position of the new item in the child list

Returns

The new item added.

See Also

insert_above()

Fl_Tree_Item ∗ Fl_Tree::insert_above (Fl_Tree_Item ∗ *above*, const char ∗ *name*)

Inserts a new item `'name'` above the specified Fl_Tree_Item `'above'`.

Example:

```
tree->add("Aaa/000");       // "000" is index 0 in Aaa's children
tree->add("Aaa/111");       // "111" is index 1 in Aaa's children
tree->add("Aaa/222");       // "222" is index 2 in Aaa's children
..
// How to use insert_above() to insert a new item above Aaa/222
Fl_Tree_Item *item = tree->find_item("Aaa/222");  // get item Aaa/222
if (item) tree->insert_above(item, "New item");   // insert new item above it
```

Parameters

in	*above*	– the item above which to insert the new item. Must not be NULL.
in	*name*	– the name of the new item

Returns

The new item added, or 0 if 'above' could not be found.

See Also

insert()

int Fl_Tree::is_close (Fl_Tree_Item ∗ *item*) const

See if the specified `'item'` is closed.

Parameters

in	*item*	– the item to be tested. Must not be NULL.

Returns

- 1 : item is closed
- 0 : item is open

int Fl_Tree::is_close (const char ∗ *path*) const

See if item specified by `'path'` is closed.

Items or submenus that themselves contain slashes ('/' or '\') should be escaped, e.g. is_close("-Holidays/12\\/25\\/2010").

Parameters

in	*path*	– the tree item's pathname (e.g. "Flintstones/Fred")

Returns

- 1 - OK: item is closed
- 0 - OK: item is open
- -1 - ERROR: item was not found

int Fl_Tree::is_hscroll_visible ( ) const

See if the horizontal scrollbar is currently visible.

Returns

1 if scrollbar visible, 0 if not.

Note

Must be using FLTK ABI 1.3.3 or higher for this to be effective.

int Fl_Tree::is_open (Fl_Tree_Item ∗ *item*) const

See if `'item'` is open.

Items that are 'open' are themselves not necessarily visible; one of the item's parents might be closed.

Parameters

`in`	*item*	– the item to be tested. Must not be NULL.

Returns

- 1 : item is open
- 0 : item is closed

int Fl_Tree::is_open (const char ∗ *path*) const

See if item specified by `'path'` is open.

Items or submenus that themselves contain slashes ('/' or '\') should be escaped, e.g. is_open("Holidays/12\\/25\\/2010").

Items that are 'open' are themselves not necessarily visible; one of the item's parents might be closed.

Parameters

`in`	*path*	– the tree item's pathname (e.g. "Flintstones/Fred")

Returns

- 1 - OK: item is open
- 0 - OK: item is closed
- -1 - ERROR: item was not found

See Also

Fl_Tree_Item::visible_r()

int Fl_Tree::is_scrollbar (Fl_Widget ∗ *w*)

See if widget `'w'` is one of the Fl_Tree widget's scrollbars.

Use this to skip over the scrollbars when walking the child() array. Example:

```
for ( int i=0; i<tree->children(); i++ ) {     // walk children
    Fl_Widget *w = tree->child(i);
    if ( tree->is_scrollbar(w) ) continue;     // skip scrollbars
    ..do work here..
}
```

Parameters

in	*w*	Widget to test

Returns

1 if `w` is a scrollbar, 0 if not.

Todo should be const

int Fl_Tree::is_selected (Fl_Tree_Item ∗ *item*) const

See if the specified `'item'` is selected.
Parameters

in	*item*	– the item to be tested. Must not be NULL.

Returns

- 1 : item selected
- 0 : item deselected

int Fl_Tree::is_selected (const char ∗ *path*)

See if item specified by `'path'` is selected.

Items or submenus that themselves contain slashes ('/' or '\') should be escaped, e.g. is_selected("Holidays/12\\/25\\/2010").
Parameters

in	*path*	– the tree item's pathname (e.g. "Flintstones/Fred")

Returns

- 1 : item selected
- 0 : item deselected
- -1 : item was not found

int Fl_Tree::is_vscroll_visible ( ) const

See if the vertical scrollbar is currently visible.

Returns

1 if scrollbar visible, 0 if not.

void Fl_Tree::item_clicked (Fl_Tree_Item ∗ *item*) `[protected]`

Set the item that was last clicked.

Should only be used by subclasses needing to change this value. Normally Fl_Tree manages this value.

Deprecated in 1.3.3 ABI – use callback_item() instead.

Fl_Tree_Item ∗ Fl_Tree::item_clicked ( )

Return the item that was last clicked.
Valid only from within the callback().

Returns

The item clicked, or 0 if none. 0 may also be used to indicate several items were clicked/changed.

Deprecated in 1.3.3 ABI – use callback_item() instead.

Fl_Tree_Item_Draw_Mode Fl_Tree::item_draw_mode ( ) const

Get the 'item draw mode' used for the tree.

Version

1.3.1 ABI feature

void Fl_Tree::item_draw_mode (Fl_Tree_Item_Draw_Mode *mode*)

Set the 'item draw mode' used for the tree to `'mode'`.
This affects how items in the tree are drawn, such as when a widget() is defined. See Fl_Tree_Item_Draw_Mode for possible values.

Version

1.3.1 ABI feature

void Fl_Tree::item_draw_mode (int *mode*)

Set the 'item draw mode' used for the tree to integer `'mode'`.
This affects how items in the tree are drawn, such as when a widget() is defined. See Fl_Tree_Item_Draw_Mode for possible values.

Version

1.3.1 ABI feature

Fl_Color Fl_Tree::item_labelbgcolor (void) const

Get the default label background color used for creating new items.
If the color is 0xffffffff, it is 'transparent'.

void Fl_Tree::item_labelbgcolor (Fl_Color *val*)

Set the default label background color used for creating new items.
A special case is made for color 0xffffffff (default) which is treated as 'transparent'. To change the background color on a per-item basis, use Fl_Tree_Item::labelbgcolor(Fl_Color)

void Fl_Tree::item_labelfgcolor (Fl_Color *val*)

Set the default label foreground color used for creating new items.
To change the foreground color on a per-item basis, use Fl_Tree_Item::labelfgcolor(Fl_Color)

void Fl_Tree::item_labelfont (Fl_Font *val*)

Set the default font face used for creating new items.

To change the font face on a per-item basis, use Fl_Tree_Item::labelfont(Fl_Font)

void Fl_Tree::item_labelsize (Fl_Fontsize *val*)

Set the default label font size used for creating new items.

To change the font size on a per-item basis, use Fl_Tree_Item::labelsize(Fl_Fontsize)

int Fl_Tree::item_pathname (char ∗ *pathname*, int *pathnamelen*, const Fl_Tree_Item ∗ *item*) const

Return `'pathname'` of size `'pathnamelen'` for the specified `'item'`.

If `'item'` is NULL, root() is used.

The tree's root will be included in the pathname if showroot() is on.

Menu items or submenus that contain slashes ('/' or '\') in their names will be escaped with a backslash. This is symmetrical with the add() function which uses the same escape pattern to set names.

Parameters

`out`	*pathname*	The string to use to return the pathname
`in`	*pathnamelen*	The maximum length of the string (including NULL). Must not be zero.
`in`	*item*	The item whose pathname is to be returned.

Returns

- 0 : OK (`pathname` returns the item's pathname)
- -1 : item not found (pathname="")
- -2 : pathname not large enough (pathname="")

See Also

find_item()

Fl_Tree_Item_Reselect_Mode Fl_Tree::item_reselect_mode ( ) const

Returns the current item re/selection mode.

Version

1.3.1 ABI feature

void Fl_Tree::item_reselect_mode (Fl_Tree_Item_Reselect_Mode *mode*)

Sets the item re/selection mode.

See Fl_Tree_Item_Reselect_Mode for possible values.

Version

1.3.1 ABI feature

int Fl_Tree::labelmarginleft ( ) const

Get the amount of white space (in pixels) that should appear to the left of the label text.

void Fl_Tree::labelmarginleft (int *val*)

Set the amount of white space (in pixels) that should appear to the left of the label text.

Fl_Tree_Item ∗ Fl_Tree::last ( )

Returns the last item in the tree.
This can be used to walk the tree in reverse, e.g.

```
for ( Fl_Tree_Item *item = tree->last(); item; item = tree->prev() )
    printf("Item: %s\n", item->label());
```

Returns

Last item in the tree, or 0 if none (tree empty).

See Also

first(), next(), last(), prev()

Fl_Tree_Item ∗ Fl_Tree::last_selected_item ( )

Returns the last selected item in the tree.
Use this to walk the tree in reverse from bottom to top looking for all the selected items, e.g.

```
// Walk tree in reverse, from bottom to top
for ( Fl_Tree_Item *i=tree->last_selected_item(); i; i=tree->
      next_selected_item(i, FL_Up) )
    printf("Selected item: %s\n", i->label());
```

Returns

The last selected item, or 0 if none.

See Also

first_selected_item(), last_selected_item(), next_selected_item()

Version

1.3.3

Fl_Tree_Item ∗ Fl_Tree::last_visible ( )

Returns the last open(), visible item in the tree.

Deprecated in 1.3.3 – use last_visible_item() instead.

Fl_Tree_Item ∗ Fl_Tree::last_visible_item ( )

Returns the last open(), visible item in the tree.

Returns

Last visible item in the tree, or 0 if none.

See Also

first_visible_item(), last_visible_item(), next_visible_item()

Version

1.3.3

void Fl_Tree::load (class Fl_Preferences & *prefs*)

Load FLTK preferences.

Read a preferences database into the tree widget.

A preferences database is a hierarchical collection of data which can be directly loaded into the tree view for inspection.

Parameters

in	*prefs*	the Fl_Preferences database

Fl_Tree_Item ∗ Fl_Tree::next (Fl_Tree_Item ∗ *item* = `0`)

Return the next item after `'item'`, or 0 if no more items.

Use this code to walk the entire tree:

```
for ( Fl_Tree_Item *i = tree->first(); i; i = tree->next(i) )
    printf("Item: %s\n", i->label());
```

Parameters

in	*item*	The item to use to find the next item. If NULL, returns 0.

Returns

Next item in tree, or 0 if at last item.

See Also

first(), next(), last(), prev()

Fl_Tree_Item ∗ Fl_Tree::next_item (Fl_Tree_Item ∗ *item*, int *dir* = FL_Down, bool *visible* = `false`)

Returns next item after `'item'` in direction `'dir'` depending on `'visible'`.

Next item will be above (if dir==FL_Up) or below (if dir==FL_Down). If `'visible'` is true, only items whose parents are open() will be returned. If `'visible'` is false, even items whose parents are close()ed will be returned.

If `item` is 0, the return value will be the result of this truth table:

```
                          visible=true            visible=false
                          -------------------     -------------
            dir=FL_Up:    last_visible_item()     last()
          dir=FL_Down:    first_visible_item()    first()
```

Example use:

```
// Walk down the tree showing open(), visible items
for ( Fl_Tree_Item *i=tree->first_visible_item(); i; i=tree->
      next_item(i, FL_Down, true) )
    printf("Item: %s\n", i->label());

// Walk up the tree showing open(), visible items
for ( Fl_Tree_Item *i=tree->last_visible_item(); i; i=tree->
      next_item(i, FL_Up, true) )
    printf("Item: %s\n", i->label());

// Walk down the tree showing all items (open or closed)
for ( Fl_Tree_Item *i=tree->first(); i; i=tree->next_item(i,
      FL_Down, false) )
    printf("Item: %s\n", i->label());

// Walk up the tree showing all items (open or closed)
for ( Fl_Tree_Item *i=tree->last(); i; i=tree->next_item(i,
      FL_Up, false) )
    printf("Item: %s\n", i->label());
```

Parameters

in	*item*	The item to use to find the next item. If NULL, returns 0.
in	*dir*	Can be FL_Up or FL_Down (default=FL_Down or 'next')
in	*visible*	true=return only open(), visible items, false=return open or closed items (default)

Returns

Next item in tree in the direction and visibility specified, or 0 if no more items of specified visibility in that direction.

See Also

first(), last(), next(),
first_visible_item(), last_visible_item(), next_visible_item(),
first_selected_item(), last_selected_item(), next_selected_item()

Version

1.3.3

Fl_Tree_Item ∗ Fl_Tree::next_selected_item (Fl_Tree_Item ∗ *item* = *0*, int *dir* = FL_Down)

Returns the next selected item above or below `'item'`, depending on `'dir'`.

If `'item'` is 0, search starts at either first() or last(), depending on `'dir'`: first() if `'dir'` is FL_-Down (default), last() if `'dir'` is FL_Up.

Use this to walk the tree looking for all the selected items, e.g.

```
// Walk down the tree (forwards)
for ( Fl_Tree_Item *i=tree->first_selected_item(); i; i=tree->
     next_selected_item(i, FL_Down) )
    printf("Item: %s\n", i->label());

// Walk up the tree (backwards)
for ( Fl_Tree_Item *i=tree->last_selected_item(); i; i=tree->
     next_selected_item(i, FL_Up) )
    printf("Item: %s\n", i->label());
```

Parameters

in	*item*	The item above or below which we'll find the next selected item. If NULL, first() is used if FL_Down, last() if FL_Up. (default=NULL)
in	*dir*	The direction to go. FL_Up for moving up the tree, FL_Down for down the tree (default)

Returns

The next selected item, or 0 if there are no more selected items.

See Also

first_selected_item(), last_selected_item(), next_selected_item()

Version

1.3.3

Fl_Tree_Item ∗ Fl_Tree::next_visible_item (Fl_Tree_Item ∗ *item*, int *dir*)

Returns next open(), visible item above (`dir==FL_Up`) or below (`dir==FL_Down`) the specified `'item'`, or 0 if no more items.

If `'item'` is 0, returns last() if `'dir'` is FL_Up, or first() if `dir` is FL_Down.

```
// Walk down the tree (forwards)
for ( Fl_Tree_Item *i=tree->first_visible_item(); i; i=tree->
      next_visible_item(i, FL_Down) )
    printf("Item: %s\n", i->label());

// Walk up the tree (backwards)
for ( Fl_Tree_Item *i=tree->last_visible_item(); i; i=tree->
      next_visible_item(i, FL_Up) )
    printf("Item: %s\n", i->label());
```

Parameters

in	*item*	The item above/below which we'll find the next visible item
in	*dir*	The direction to search. Can be FL_Up or FL_Down.

Returns

The item found, or 0 if there's no visible items above/below the specified `item`.

Version

1.3.3

int Fl_Tree::open (Fl_Tree_Item ∗ *item*, int *docallback* = *1*)

Open the specified `'item'`.

This causes the item's children (if any) to be shown.

Invokes the callback depending on the value of optional parameter `'docallback'`.

Handles calling redraw() if anything changed.

The callback can use callback_item() and callback_reason() respectively to determine the item changed and the reason the callback was called.

Parameters

in	*item*	– the item to be opened. Must not be NULL.
in	*docallback*	– A flag that determines if the callback() is invoked or not: • 0 - callback() is not invoked • 1 - callback() is invoked if item changed (default), callback_reason() will be FL_TREE_REASON_OPENED

Returns

- 1 – item was opened
- 0 – item was already open, no change

See Also

open(), close(), is_open(), is_close(), callback_item(), callback_reason()

int Fl_Tree::open (const char ∗ *path*, int *docallback* = `1`)

Opens the item specified by `'path'`.

This causes the item's children (if any) to be shown.

Invokes the callback depending on the value of optional parameter `'docallback'`.

Handles calling redraw() if anything changed.

Items or submenus that themselves contain slashes ('/' or '\') should be escaped, e.g. open("Holidays/12\\/25\\/2010").

The callback can use callback_item() and callback_reason() respectively to determine the item changed and the reason the callback was called.

Parameters

`in`	*path*	– the tree item's pathname (e.g. "Flintstones/Fred")
`in`	*docallback*	– A flag that determines if the callback() is invoked or not: • 0 - callback() is not invoked • 1 - callback() is invoked if item changed (default), callback_-reason() will be FL_TREE_REASON_OPENED

Returns

- 1 – OK: item opened
- 0 – OK: item was already open, no change
- -1 – ERROR: item was not found

See Also

open(), close(), is_open(), is_close(), callback_item(), callback_reason()

void Fl_Tree::open_toggle (Fl_Tree_Item ∗ *item*, int *docallback* = `1`)

Toggle the open state of `'item'`.

Invokes the callback depending on the value of optional parameter `'docallback'`.

Handles calling redraw() if anything changed.

The callback can use callback_item() and callback_reason() respectively to determine the item changed and the reason the callback was called.

Parameters

`in`	*item*	– the item whose open state is to be toggled. Must not be NULL.
`in`	*docallback*	– A flag that determines if the callback() is invoked or not: • 0 - callback() is not invoked • 1 - callback() is invoked (default), callback_reason() will be either FL_TREE_REASON_OPENED or FL_TREE_REASON_CLOSE-D

See Also

open(), close(), is_open(), is_close(), callback_item(), callback_reason()

Fl_Image ∗ Fl_Tree::openicon ( ) const

Returns the icon to be used as the 'open' icon.

If none was set, the internal default is returned, a simple '[+]' icon.

void Fl_Tree::openicon (Fl_Image ∗ *val*)

Sets the icon to be used as the 'open' icon.
This overrides the built in default '[+]' icon.

Parameters

in	*val*	– The new image, or zero to use the default [+] icon.

Fl_Tree_Item ∗ Fl_Tree::prev (Fl_Tree_Item ∗ *item* = `0`)

Return the previous item before `'item'`, or 0 if no more items.
This can be used to walk the tree in reverse, e.g.

```
for ( Fl_Tree_Item *item = tree->first(); item; item = tree->
      prev(item) )
    printf("Item: %s\n", item->label());
```

Parameters

in	*item*	The item to use to find the previous item. If NULL, returns 0.

Returns

Previous item in tree, or 0 if at first item.

See Also

first(), next(), last(), prev()

void Fl_Tree::recalc_tree ( )

Schedule tree to recalc the entire tree size.

Note

Must be using FLTK ABI 1.3.3 or higher for this to be effective.

int Fl_Tree::remove (Fl_Tree_Item ∗ *item*)

Remove the specified `'item'` from the tree.
`item` may not be NULL. If it has children, all those are removed too. If item being removed has focus, no item will have focus.

Returns

0 if done, -1 if 'item' not found.

void Fl_Tree::resize (int *X*, int *Y*, int *W*, int *H*) `[virtual]`

Resizes the Fl_Group widget and all of its children.
The Fl_Group widget first resizes itself, and then it moves and resizes all its children according to the rules documented for Fl_Group::resizable(Fl_Widget∗)

See Also

Fl_Group::resizable(Fl_Widget∗)
Fl_Group::resizable()
Fl_Widget::resize(int,int,int,int)

Reimplemented from Fl_Group.

void Fl_Tree::root (Fl_Tree_Item ∗ *newitem*)

Sets the root item to `'newitem'`.

If a root item already exists, clear() is called first to clear it before replacing it with newitem.Use this to install a custom item (derived from Fl_Tree_Item) as the root of the tree. This allows the derived class to implement custom drawing by overriding Fl_Tree_Item::draw_item_content().

Version

1.3.3

void Fl_Tree::root_label (const char ∗ *new_label*)

Set the label for the root item to `'new_label'`.

Makes an internally managed copy of 'new_label'.

int Fl_Tree::scrollbar_size ( ) const

Gets the default size of scrollbars' troughs for this widget in pixels.

If this value is zero (default), this widget will use the global Fl::scrollbar_size() value as the scrollbar's width.

Returns

Scrollbar size in pixels, or 0 if the global Fl::scrollbar_size() is being used.

See Also

Fl::scrollbar_size(int)

void Fl_Tree::scrollbar_size (int *size*)

Sets the pixel size of the scrollbars' troughs to `'size'` for this widget, in pixels.

Normally you should not need this method, and should use the global Fl::scrollbar_size(int) instead to manage the size of ALL your widgets' scrollbars. This ensures your application has a consistent UI, and is the default behavior. Normally this is what you want.

Only use this method if you really need to override just THIS instance of the widget's scrollbar size. (This need should be rare.)

Setting `size` to the special value of 0 causes the widget to track the global Fl::scrollbar_size(), which is the default.

Parameters

`in`	*size*	Sets the scrollbar size in pixels. If 0 (default), scrollbar size tracks the global Fl::scrollbar_size()

See Also

Fl::scrollbar_size()

int Fl_Tree::select (Fl_Tree_Item ∗ *item*, int *docallback* = 1)

Select the specified `'item'`.

Use 'deselect()' to deselect it.

Invokes the callback depending on the value of optional parameter `docallback`.

Handles calling redraw() if anything changed.

The callback can use callback_item() and callback_reason() respectively to determine the item changed and the reason the callback was called.

Parameters

`in`	*item*	– the item to be selected. Must not be NULL.
`in`	*docallback*	– A flag that determines if the callback() is invoked or not: • 0 - the callback() is not invoked • 1 - the callback() is invoked if item changed state, callback_reason() will be FL_TREE_REASON_SELECTED

Returns

- 1 - item's state was changed
- 0 - item was already selected, no change was made

int Fl_Tree::select (const char ∗ ***path*****, int *docallback* = `1`)**

Select the item specified by `'path'`.

Invokes the callback depending on the value of optional parameter `'docallback'`.

Handles calling redraw() if anything changed.

Items or submenus that themselves contain slashes ('/' or '\') should be escaped, e.g. select("Holidays/12\\/25\\/2010").

The callback can use callback_item() and callback_reason() respectively to determine the item changed and the reason the callback was called.

Parameters

`in`	*path*	– the tree item's pathname (e.g. "Flintstones/Fred")
`in`	*docallback*	– A flag that determines if the callback() is invoked or not: • 0 - the callback() is not invoked • 1 - the callback() is invoked if item changed state (default), callback_reason() will be FL_TREE_REASON_SELECTED

Returns

- 1 : OK: item's state was changed
- 0 : OK: item was already selected, no change was made
- -1 : ERROR: item was not found

int Fl_Tree::select_all (Fl_Tree_Item ∗ ***item* = `0`, int *docallback* = `1`)**

Select `'item'` and all its children.

If item is NULL, first() is used.

Invokes the callback depending on the value of optional parameter `'docallback'`.

Handles calling redraw() if anything changed.

The callback can use callback_item() and callback_reason() respectively to determine the item changed and the reason the callback was called.

Parameters

`in`	*item*	The item that will be selected (along with all its children). If NULL, first() is used.
`in`	*docallback*	– A flag that determines if the callback() is invoked or not: • 0 - the callback() is not invoked • 1 - the callback() is invoked for each item that changed state (default), callback_reason() will be FL_TREE_REASON_SELECTED

Returns

Count of how many items were actually changed to the selected state.

int Fl_Tree::select_only (Fl_Tree_Item ∗ *selitem*, int *docallback* = `1`)

Select only the specified item, deselecting all others that might be selected.

If `'selitem'` is 0, first() is used.

Invokes the callback depending on the value of optional parameter `'docallback'`.

Handles calling redraw() if anything changed.

The callback can use callback_item() and callback_reason() respectively to determine the item changed and the reason the callback was called.

Parameters

`in`	*selitem*	The item to be selected. If NULL, first() is used.
`in`	*docallback*	– A flag that determines if the callback() is invoked or not: • 0 - the callback() is not invoked • 1 - the callback() is invoked for each item that changed state (default), callback_reason() will be either FL_TREE_REASON_SELECTED or FL_TREE_REASON_DESELECTED

Returns

The number of items whose selection states were changed, if any.

void Fl_Tree::select_toggle (Fl_Tree_Item ∗ *item*, int *docallback* = `1`)

Toggle the select state of the specified `'item'`.

Invokes the callback depending on the value of optional parameter `'docallback'`.

Handles calling redraw() if anything changed.

The callback can use callback_item() and callback_reason() respectively to determine the item changed and the reason the callback was called.

Parameters

`in`	*item*	– the item to be selected. Must not be NULL.
`in`	*docallback*	– A flag that determines if the callback() is invoked or not: • 0 - the callback() is not invoked • 1 - the callback() is invoked (default), callback_reason() will be either FL_TREE_REASON_SELECTED or FL_TREE_REASON_DESELECTED

Fl_Boxtype Fl_Tree::selectbox ( ) const

Sets the style of box used to draw selected items.

This is an fltk Fl_Boxtype. The default is influenced by FLTK's current Fl::scheme()

void Fl_Tree::selectbox (Fl_Boxtype *val*)

Gets the style of box used to draw selected items.

This is an fltk Fl_Boxtype. The default is influenced by FLTK's current Fl::scheme()

Fl_Tree_Select Fl_Tree::selectmode ( ) const

Gets the tree's current selection mode.

See Fl_Tree_Select for possible values.

void Fl_Tree::selectmode (Fl_Tree_Select *val*)

Sets the tree's selection mode.

See Fl_Tree_Select for possible values.

void Fl_Tree::set_item_focus (Fl_Tree_Item ∗ *item*)

Set the item that currently should have keyboard focus.

Handles calling redraw() to update the focus box (if it is visible).

Parameters

`in`	*item*	The item that should take focus. If NULL, none will have focus.

void Fl_Tree::show_item (Fl_Tree_Item ∗ *item*, int *yoff*)

Adjust the vertical scrollbar so that `'item'` is visible `'yoff'` pixels from the top of the Fl_Tree widget's display.

For instance, yoff=0 will position the item at the top.

If yoff is larger than the vertical scrollbar's limit, the value will be clipped. So if yoff=100, but scrollbar's max is 50, then 50 will be used.

Parameters

`in`	*item*	The item to be shown. If NULL, first() is used.
`in`	*yoff*	The pixel offset from the top for the displayed position.

See Also

show_item_top(), show_item_middle(), show_item_bottom()

void Fl_Tree::show_item (Fl_Tree_Item ∗ *item*)

Adjust the vertical scrollbar to show `'item'` at the top of the display IF it is currently off-screen (for instance show_item_top()).

If it is already on-screen, no change is made.

Parameters

`in`	*item*	The item to be shown. If NULL, first() is used.

See Also

show_item_top(), show_item_middle(), show_item_bottom()

void Fl_Tree::show_item_bottom (Fl_Tree_Item ∗ *item*)

Adjust the vertical scrollbar so that `'item'` is at the bottom of the display.
Parameters

`in`	*item*	The item to be shown. If NULL, first() is used.

void Fl_Tree::show_item_middle (Fl_Tree_Item ∗ *item*)

Adjust the vertical scrollbar so that `'item'` is in the middle of the display.
Parameters

`in`	*item*	The item to be shown. If NULL, first() is used.

void Fl_Tree::show_item_top (Fl_Tree_Item ∗ *item*)

Adjust the vertical scrollbar so that `'item'` is at the top of the display.
Parameters

`in`	*item*	The item to be shown. If NULL, first() is used.

void Fl_Tree::show_self ( )

Print the tree as 'ascii art' to stdout.
Used mainly for debugging.

Todo should be const

Version

1.3.0

int Fl_Tree::showcollapse ( ) const

Returns 1 if the collapse icon is enabled, 0 if not.

See Also

showcollapse(int)

void Fl_Tree::showcollapse (int *val*)

Set if we should show the collapse icon or not.
If collapse icons are disabled, the user will not be able to interactively collapse items in the tree, unless the application provides some other means via open() and close().

Parameters

in	*val*	1: shows collapse icons (default), 0: hides collapse icons.

void Fl_Tree::showroot (int *val*)

Set if the root item should be shown or not.
Parameters

in	*val*	1 – show the root item (default) 0 – hide the root item.

Fl_Tree_Sort Fl_Tree::sortorder ( ) const

Set the default sort order used when items are added to the tree.
See Fl_Tree_Sort for possible values.

Fl_Image ∗ Fl_Tree::usericon ( ) const

Returns the Fl_Image being used as the default user icon for all newly created items.
Returns zero if no icon has been set, which is the default.

void Fl_Tree::usericon (Fl_Image ∗ *val*)

Sets the Fl_Image to be used as the default user icon for all newly created items.
If you want to specify user icons on a per-item basis, use Fl_Tree_Item::usericon() instead.
Parameters

in	*val*	– The new image to be used, or zero to disable user icons.

int Fl_Tree::usericonmarginleft ( ) const

Get the amount of white space (in pixels) that should appear to the left of the usericon.

void Fl_Tree::usericonmarginleft (int *val*)

Set the amount of white space (in pixels) that should appear to the left of the usericon.

int Fl_Tree::vposition ( ) const

Returns the vertical scroll position as a pixel offset.
The position returned is how many pixels of the tree are scrolled off the top edge of the screen.

See Also

vposition(int), hposition(), hposition(int)

void Fl_Tree::vposition (int *pos*)

Sets the vertical scroll offset to position `'pos'`.
The position is how many pixels of the tree are scrolled off the top edge of the screen.

Parameters

`in`	*pos*	The vertical position (in pixels) to scroll the tree to.

See Also

vposition(), hposition(), hposition(int)

int Fl_Tree::widgetmarginleft ( ) const

Get the amount of white space (in pixels) that should appear to the left of the child fltk widget (if any).

void Fl_Tree::widgetmarginleft (int *val*)

Set the amount of white space (in pixels) that should appear to the left of the child fltk widget (if any).

The documentation for this class was generated from the following files:

- Fl_Tree.H
- Fl_Tree.cxx

31.144 Fl_Tree_Item Class Reference

Tree widget item.

`#include <Fl_Tree_Item.H>`

Public Member Functions

- void activate (int val=1)

  Change the item's activation state to the optionally specified 'val'.
- Fl_Tree_Item ∗ add (const Fl_Tree_Prefs &prefs, const char ∗new_label, Fl_Tree_Item ∗newitem)

  Add `'item'` as immediate child with `'new_label'` and defaults from `'prefs'`.
- Fl_Tree_Item ∗ add (const Fl_Tree_Prefs &prefs, const char ∗new_label)

  Add a new child to this item with the name `'new_label'` and defaults from `'prefs'`.
- Fl_Tree_Item ∗ add (const Fl_Tree_Prefs &prefs, char ∗∗arr, Fl_Tree_Item ∗newitem)

  Descend into path specified by `'arr'` and add `'newitem'` there.
- Fl_Tree_Item ∗ add (const Fl_Tree_Prefs &prefs, char ∗∗arr)

  Descend into the path specified by `'arr'`, and add a new child there.
- Fl_Tree_Item ∗ child (int index)

  Return the child item for the given 'index'.
- const Fl_Tree_Item ∗ child (int t) const

  Return the const child item for the given 'index'.
- int children () const

  Return the number of children this item has.
- void clear_children ()

  Clear all the children for this item.
- void close ()

  Close this item and all its children.
- void deactivate ()

  Deactivate the item; the callback() won't be invoked when clicked.
- Fl_Tree_Item ∗ deparent (int index)

Deparent child at index position `'pos'`.

- int depth () const

  Returns how many levels deep this item is in the hierarchy.
- void deselect ()

  Disable the item's selection state.
- int deselect_all ()

  Deselect item and all its children.
- void draw (int X, int &Y, int W, Fl_Tree_Item ∗itemfocus, int &tree_item_xmax, int lastchild=1, int render=1)

  Draw this item and its children.
- virtual int draw_item_content (int render)

  Draw the item content.
- int event_on_collapse_icon (const Fl_Tree_Prefs &prefs) const

  Was the event on the 'collapse' button of this item?
- int event_on_label (const Fl_Tree_Prefs &prefs) const

  Was event on the label() of this item?
- int find_child (const char ∗name)

  Return the index of the immediate child of this item that has the label `'name'`.
- int find_child (Fl_Tree_Item ∗item)

  Find the index number for the specified `'item'` *in the current item's list of children.*
- const Fl_Tree_Item ∗ find_child_item (const char ∗name) const

  Return the /immediate/ child of current item that has the label `'name'`.
- Fl_Tree_Item ∗ find_child_item (const char ∗name)

  Non-const version of Fl_Tree_Item::find_child_item(const char ∗name) const.
- const Fl_Tree_Item ∗ find_child_item (char ∗∗arr) const

  Find child item by descending array `'arr'` *of names.*
- Fl_Tree_Item ∗ find_child_item (char ∗∗arr)

  Non-const version of Fl_Tree_Item::find_child_item(char ∗∗arr) const.
- const Fl_Tree_Item ∗ find_clicked (const Fl_Tree_Prefs &prefs, int yonly=0) const

  Find the item that the last event was over.
- Fl_Tree_Item ∗ find_clicked (const Fl_Tree_Prefs &prefs, int yonly=0)

  Non-const version of Fl_Tree_Item::find_clicked(const Fl_Tree_Prefs&,int) const.
- const Fl_Tree_Item ∗ find_item (char ∗∗arr) const

  Find item by descending array of `'names'`.
- Fl_Tree_Item ∗ find_item (char ∗∗arr)

  Non-const version of Fl_Tree_Item::find_item(char ∗∗names) const.
- Fl_Tree_Item (const Fl_Tree_Prefs &prefs)

  Constructor.
- Fl_Tree_Item (Fl_Tree ∗tree)

  Constructor.
- Fl_Tree_Item (const Fl_Tree_Item ∗o)

  Copy constructor.
- int h () const

  The item's height.
- int has_children () const

  See if this item has children.
- Fl_Tree_Item ∗ insert (const Fl_Tree_Prefs &prefs, const char ∗new_label, int pos=0)

Insert a new item named `'new_label'` *into current item's children at a specified position* `'pos'`*.*

- Fl_Tree_Item ∗ insert_above (const Fl_Tree_Prefs &prefs, const char ∗new_label)

  Insert a new item named `'new_label'` *above this item.*

- char is_activated () const

  See if the item is activated.

- char is_active () const

  See if the item is activated. Alias for is_activated().

- int is_close () const

  See if the item is 'closed'.

- int is_open () const

  See if the item is 'open'.

- int is_root () const

  Is this item the root of the tree?

- char is_selected () const

  See if the item is selected.

- int is_visible () const

  See if the item is visible.

- void label (const char ∗val)

  Set the label to `'name'`*.*

- const char ∗ label () const

  Return the label.

- int label_h () const

  The item's label height.

- int label_w () const

  The item's maximum label width to right edge of Fl_Tree's inner width within scrollbars.

- int label_x () const

  The item's label x position relative to the window.

- int label_y () const

  The item's label y position relative to the window.

- void labelbgcolor (Fl_Color val)

  Set item's label background color.

- Fl_Color labelbgcolor () const

  Return item's label background text color.

- void labelcolor (Fl_Color val)

  Set item's label text color. Alias for labelfgcolor(Fl_Color)).

- Fl_Color labelcolor () const

  Return item's label text color. Alias for labelfgcolor() const).

- void labelfgcolor (Fl_Color val)

  Set item's label foreground text color.

- Fl_Color labelfgcolor () const

  Return item's label foreground text color.

- void labelfont (Fl_Font val)

  Set item's label font face.

- Fl_Font labelfont () const

  Get item's label font face.

- void labelsize (Fl_Fontsize val)

Set item's label font size.

- Fl_Fontsize labelsize () const

  Get item's label font size.
- int move (int to, int from)

  Move the item 'from' to sibling position of 'to'.
- int move (Fl_Tree_Item ∗item, int op=0, int pos=0)

  Move the current item above/below/into the specified 'item', where `'op'` determines the type of move:
- int move_above (Fl_Tree_Item ∗item)

  Move the current item above the specified 'item'.
- int move_below (Fl_Tree_Item ∗item)

  Move the current item below the specified 'item'.
- int move_into (Fl_Tree_Item ∗item, int pos=0)

  Parent the current item as a child of the specified `'item'`.
- Fl_Tree_Item ∗ next ()

  Return the next item in the tree.
- Fl_Tree_Item ∗ next_displayed (Fl_Tree_Prefs &prefs)

  Same as next_visible().
- Fl_Tree_Item ∗ next_sibling ()

  Return this item's next sibling.
- Fl_Tree_Item ∗ next_visible (Fl_Tree_Prefs &prefs)

  Return the next open(), visible() item.
- void open ()

  Open this item and all its children.
- void open_toggle ()

  Toggle the item's open/closed state.
- Fl_Tree_Item ∗ parent ()

  Return the parent for this item. Returns NULL if we are the root.
- const Fl_Tree_Item ∗ parent () const

  Return the const parent for this item. Returns NULL if we are the root.
- void parent (Fl_Tree_Item ∗val)

  Set the parent for this item.
- const Fl_Tree_Prefs & prefs () const

  Return the parent tree's prefs.
- Fl_Tree_Item ∗ prev ()

  Return the previous item in the tree.
- Fl_Tree_Item ∗ prev_displayed (Fl_Tree_Prefs &prefs)

  Same as prev_visible().
- Fl_Tree_Item ∗ prev_sibling ()

  Return this item's previous sibling.
- Fl_Tree_Item ∗ prev_visible (Fl_Tree_Prefs &prefs)

  Return the previous open(), visible() item.
- int remove_child (Fl_Tree_Item ∗item)

  Remove `'item'` from the current item's children.
- int remove_child (const char ∗new_label)

  Remove immediate child (and its children) by its label `'name'`.
- int reparent (Fl_Tree_Item ∗newchild, int index)

Reparent specified item as a child of ourself at position `'pos'`*.*

- Fl_Tree_Item ∗ replace (Fl_Tree_Item ∗new_item)

  Replace the current item with a new item.
- Fl_Tree_Item ∗ replace_child (Fl_Tree_Item ∗olditem, Fl_Tree_Item ∗newitem)

  Replace existing child `'olditem'` *with* `'newitem'`*.*
- void select (int val=1)

  Change the item's selection state to the optionally specified 'val'.
- int select_all ()

  Select item and all its children.
- void select_toggle ()

  Toggle the item's selection state.
- void show_self (const char ∗indent="") const

  Print the tree as 'ascii art' to stdout.
- void swap_children (int ax, int bx)

  Swap two of our children, given two child index values `'ax'` *and* `'bx'`*.*
- int swap_children (Fl_Tree_Item ∗a, Fl_Tree_Item ∗b)

  Swap two of our immediate children, given item pointers.
- const Fl_Tree ∗ tree () const

  Return the tree for this item.
- Fl_Tree ∗ tree ()

  Return the tree for this item.
- void update_prev_next (int index)

  Update our _prev_sibling and _next_sibling pointers to point to neighbors given `index` *as being our current position in the parent's item array.*
- void user_data (void ∗data)

  Set a user-data value for the item.
- void ∗ user_data () const

  Retrieve the user-data value that has been assigned to the item.
- void userdeicon (Fl_Image ∗val)

  Set the usericon to draw when the item is deactivated.
- Fl_Image ∗ userdeicon () const

  Return the deactivated version of the user icon, if any.
- void usericon (Fl_Image ∗val)

  Set the item's user icon to an Fl_Image.
- Fl_Image ∗ usericon () const

  Get the item's user icon as an Fl_Image. Returns '0' if disabled.
- int visible () const

  See if the item is visible. Alias for is_visible().
- int visible_r () const

  See if item and all its parents are open() and visible().
- int w () const

  The entire item's width to right edge of Fl_Tree's inner width within scrollbars.
- void widget (Fl_Widget ∗val)

  Assign an FLTK widget to this item.
- Fl_Widget ∗ widget () const

  Return FLTK widget assigned to this item.
- int x () const

The item's x position relative to the window.

- int y () const

  The item's y position relative to the window.

Protected Member Functions

- void **_Init** (const Fl_Tree_Prefs &prefs, Fl_Tree ∗tree)
- int calc_item_height (const Fl_Tree_Prefs &prefs) const

  Return the item's 'visible' height.
- void draw_horizontal_connector (int x1, int x2, int y, const Fl_Tree_Prefs &prefs)

  Internal: Horizontal connector line based on preference settings.
- void draw_vertical_connector (int x, int y1, int y2, const Fl_Tree_Prefs &prefs)

  Internal: Vertical connector line based on preference settings.
- Fl_Color drawbgcolor () const

  Returns the recommended background color used for drawing this item.
- Fl_Color drawfgcolor () const

  Returns the recommended foreground color used for drawing this item.
- void hide_widgets ()

  Internal: Hide the FLTK widget() for this item and all children.
- int is_flag (unsigned short val) const

  See if flag set. Returns 0 or 1.
- void recalc_tree ()

  Call this when our geometry is changed.
- void set_flag (unsigned short flag, int val)

  Set a flag to an on or off value. val is 0 or 1.
- void show_widgets ()

  Internal: Show the FLTK widget() for this item and all children.

31.144.1 Detailed Description

Tree widget item.

This class is a single tree item, and manages all of the item's attributes. Fl_Tree_Item is used by Fl_Tree, which is comprised of many instances of Fl_Tree_Item.

Fl_Tree_Item is hierarchical; it dynamically manages an Fl_Tree_Item_Array of children that are themselves instances of Fl_Tree_Item. Each item can have zero or more children. When an item has children, close() and open() can be used to hide or show them.

Items have their own attributes; font size, face, color. Items maintain their own hierarchy of children.

When you make changes to items, you'll need to tell the tree to redraw() for the changes to show up.

New 1.3.3 ABI feature: You can define custom items by either adding a custom widget to the item with Fl_Tree_Item::widget(), or override the draw_item_content() method if you want to just redefine how the label is drawn.

The following shows the Fl_Tree_Item's dimensions, useful when overriding the draw_item_content() method:

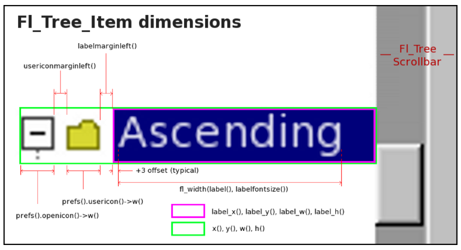


Figure 31.48: Fl_Tree_Item's internal dimensions.

31.144.2 Constructor & Destructor Documentation

Fl_Tree_Item::Fl_Tree_Item (const Fl_Tree_Prefs & *prefs*)

Constructor.

Makes a new instance of Fl_Tree_Item using defaults from `'prefs'`.

Deprecated in 1.3.3 ABI – you must use Fl_Tree_Item(Fl_Tree∗) for proper horizontal scrollbar behavior.

Fl_Tree_Item::Fl_Tree_Item (Fl_Tree ∗ *tree*)

Constructor.

Makes a new instance of Fl_Tree_Item for `'tree'`.

This must be used instead of the older, deprecated Fl_Tree_Item(Fl_Tree_Prefs) constructor for proper horizontal scrollbar calculation.

Version

1.3.3 ABI feature

31.144.3 Member Function Documentation

void Fl_Tree_Item::activate (int *val* = *1*) `[inline]`

Change the item's activation state to the optionally specified 'val'.

When deactivated, the item will be 'grayed out'; the callback() won't be invoked if the user clicks on the label. If a widget() is associated with the item, its activation state will be changed as well.

If 'val' is not specified, the item will be activated.

Fl_Tree_Item ∗ Fl_Tree_Item::add (const Fl_Tree_Prefs & *prefs*, const char ∗ *new_label*, Fl_Tree_Item ∗ *item*)

Add `'item'` as immediate child with `'new_label'` and defaults from `'prefs'`.

If `'item'` is NULL, a new item is created. An internally managed copy is made of the label string. Adds the item based on the value of prefs.sortorder().

Returns

the item added

Version

1.3.3

Fl_Tree_Item ∗ Fl_Tree_Item::add (const Fl_Tree_Prefs & *prefs*, const char ∗ *new_label*)

Add a new child to this item with the name `'new_label'` and defaults from `'prefs'`.

An internally managed copy is made of the label string. Adds the item based on the value of prefs.-sortorder().

Returns

the item added

Version

1.3.0 release

Fl_Tree_Item ∗ Fl_Tree_Item::add (const Fl_Tree_Prefs & *prefs*, char ∗∗ *arr*, Fl_Tree_Item ∗ *newitem*)

Descend into path specified by `'arr'` and add `'newitem'` there.

Should be used only by Fl_Tree's internals. If item is NULL, a new item is created. Adds the item based on the value of prefs.sortorder().

Returns

the item added.

Version

1.3.3 ABI feature

Fl_Tree_Item ∗ Fl_Tree_Item::add (const Fl_Tree_Prefs & *prefs*, char ∗∗ *arr*)

Descend into the path specified by `'arr'`, and add a new child there.

Should be used only by Fl_Tree's internals. Adds the item based on the value of prefs.sortorder().

Returns

the item added.

Version

1.3.0 release

int Fl_Tree_Item::calc_item_height (const Fl_Tree_Prefs & *prefs*) const `[protected]`

Return the item's 'visible' height.

Takes into account the item's:

- visibility (if !is_visible(), returns 0)
- labelfont() height: if label() != NULL
- widget() height: if widget() != NULL
- openicon() height (if not NULL)
- usericon() height (if not NULL) Does NOT include Fl_Tree::linespacing();

  Returns

  maximum pixel height

const Fl_Tree_Item ∗ Fl_Tree_Item::child (int *t*) const

Return the const child item for the given 'index'.
Return const child item for the specified 'index'.

void Fl_Tree_Item::deactivate ( ) `[inline]`

Deactivate the item; the callback() won't be invoked when clicked.
Same as activate(0)

Fl_Tree_Item ∗ Fl_Tree_Item::deparent (int *pos*)

Deparent child at index position `'pos'`.
This creates an "orphaned" item that is still allocated, but has no parent or siblings. Normally the caller would want to immediately reparent the orphan elsewhere.
A successfully orphaned item will have its parent() and prev_sibling()/next_sibling() set to NULL.

Returns

- pointer to orphaned item on success
- NULL on error (could not deparent the item)

int Fl_Tree_Item::depth ( ) const

Returns how many levels deep this item is in the hierarchy.
For instance; root has a depth of zero, and its immediate children would have a depth of 1, and so on. Use e.g. for determining the horizontal indent of this item during drawing.

int Fl_Tree_Item::deselect_all ( ) `[inline]`

Deselect item and all its children.
Returns count of how many items were in the 'selected' state, ie. how many items were "changed".

void Fl_Tree_Item::draw (int *X*, int & *Y*, int *W*, Fl_Tree_Item ∗ *itemfocus*, int & *tree_item_xmax*, int *lastchild* = `1`, int *render* = `1`)

Draw this item and its children.
Parameters

`in`	*X*	Horizontal position for item being drawn
`in,out`	*Y*	Vertical position for item being drawn, returns new position for next item
`in`	*W*	Recommended width for item
`in`	*itemfocus*	The tree's current focus item (if any)
`in,out`	*tree_item_xmax*	The tree's running xmax (right-most edge so far). Mainly used by parent tree when render==0 to calculate tree's max width.
`in`	*lastchild*	Is this item the last child in a subtree?
`in`	*render*	Whether or not to render the item: 0: no rendering, just calculate size w/out drawing. 1: render item as well as size calc

Version

1.3.3 ABI feature: modified parameters

void Fl_Tree_Item::draw_horizontal_connector (int *x1*, int *x2*, int *y*, const Fl_Tree_Prefs & *prefs*) `[protected]`

Internal: Horizontal connector line based on preference settings.

Parameters

in	*x1*	The left hand X position of the horizontal connector
in	*x2*	The right hand X position of the horizontal connector
in	*y*	The vertical position of the horizontal connector
in	*prefs*	The Fl_Tree prefs

int Fl_Tree_Item::draw_item_content (int *render*) [virtual]

Draw the item content.

This method can be overridden to implement custom drawing by filling the label_[xywh]() area with content.

A minimal example of how to override draw_item_content() and draw just a normal item's background and label ourselves:

```
class MyTreeItem : public Fl_Tree_Item {
public:
    MyTreeItem() { }
    ~MyTreeItem() { }
    // DRAW OUR CUSTOM CONTENT FOR THE ITEM
    int draw_item_content(int render) {
      // Our item's dimensions + text content
      int X=label_x(), Y=label_y(), W=label_w(), H=label_h();
      const char *text = label() ? label() : "";
      // Rendering? Do any drawing that's needed
      if ( render ) {
        // Draw bg -- a filled rectangle
        fl_color(drawbgcolor()); fl_rectf(X,Y,W,H);
        // Draw label
        fl_font(labelfont(), labelsize());      // use item's label font/size
        fl_color(drawfgcolor());                // use recommended fg color
        fl_draw(text, X,Y,W,H, FL_ALIGN_LEFT);  // draw the item's label
      }
      // Rendered or not, we must calculate content's max X position
      int lw=0, lh=0;
      fl_measure(text, lw, lh);                 // get width of label text
      return X + lw;                             // return X + label width
   }
};
```

You can draw anything you want inside draw_item_content() using any of the fl_draw.H functions, as long as it's within the label's xywh area.

To add instances of your custom item to the tree, you can use:

```
// Example #1: using add()
MyTreeItem *bart = new MyTreeItem(..);  // class derived from Fl_Tree_Item
tree->add("/Simpsons/Bart", bart);      // Add item as /Simpsons/Bart
```

..or you can insert or replace existing items:

```
// Example #2: using replace()
MyTreeItem *marge = new MyTreeItem(..); // class derived from Fl_Tree_Item
item = tree->add("/Simpsons/Marge");    // create item
item->replace(mi);                      // replace it with our own
```

Parameters

in	*render*	Whether we should render content (1), or just tally the geometry (0). Fl_Tree may want only to find the widest item in the tree for scrollbar calculations.

Returns

the right-most X coordinate, or 'xmax' of content we drew, i.e. the "scrollable" content. The tree uses the largest xmax to determine the maximum width of the tree's content (needed for e.g. computing the horizontal scrollbar's size).

Version

1.3.3 ABI feature

void Fl_Tree_Item::draw_vertical_connector (int *x,* int *y1,* int *y2,* const Fl_Tree_Prefs & *prefs*) [protected]

Internal: Vertical connector line based on preference settings.
Parameters

in	*x*	The x position of the vertical connector
in	*y1*	The top of the vertical connector
in	*y2*	The bottom of the vertical connector
in	*prefs*	The Fl_Tree prefs

Fl_Color Fl_Tree_Item::drawbgcolor ( ) const [protected]

Returns the recommended background color used for drawing this item.

See Also

draw_item_content()

Version

1.3.3 ABI

Fl_Color Fl_Tree_Item::drawfgcolor ( ) const [protected]

Returns the recommended foreground color used for drawing this item.

See Also

draw_item_content()

Version

1.3.3 ABI ABI

int Fl_Tree_Item::find_child (const char ∗ *name*)

Return the index of the immediate child of this item that has the label `'name'`.
Returns

index of found item, or -1 if not found.

Version

1.3.0 release

int Fl_Tree_Item::find_child (Fl_Tree_Item ∗ *item*)

Find the index number for the specified `'item'` in the current item's list of children.

Returns

the index, or -1 if not found.

const Fl_Tree_Item ∗ Fl_Tree_Item::find_child_item (const char ∗ *name*) const

Return the /immediate/ child of current item that has the label `'name'`.

Returns

const found item, or 0 if not found.

Version

1.3.3

const Fl_Tree_Item ∗ Fl_Tree_Item::find_child_item (char ∗∗ *arr*) const

Find child item by descending array `'arr'` of names.
Does not include self in search. Only Fl_Tree should need this method.

Returns

item, or 0 if not found

Version

1.3.0 release

const Fl_Tree_Item ∗ Fl_Tree_Item::find_clicked (const Fl_Tree_Prefs & *prefs*, int *yonly* = `0`) const

Find the item that the last event was over.
If `'yonly'` is 1, only check event's y value, don't care about x.
Parameters

`in`	*prefs*	The parent tree's Fl_Tree_Prefs
`in`	*yonly*	– 0: check both event's X and Y values. – 1: only check event's Y value, don't care about X.

Returns

pointer to clicked item, or NULL if none found

Version

1.3.3 ABI feature

const Fl_Tree_Item ∗ Fl_Tree_Item::find_item (char ∗∗ *names*) const

Find item by descending array of `'names'`.
Includes self in search. Only Fl_Tree should need this method. Use Fl_Tree::find_item() instead.

Returns

const item, or 0 if not found

void Fl_Tree_Item::hide_widgets ( ) `[protected]`

Internal: Hide the FLTK widget() for this item and all children.
Used by close() to hide widgets.

Fl_Tree_Item ∗ Fl_Tree_Item::insert (const Fl_Tree_Prefs & *prefs*, const char ∗ *new_label*, int *pos* = `0`)

Insert a new item named `'new_label'` into current item's children at a specified position `'pos'`.

Returns

the new item inserted.

Fl_Tree_Item ∗ Fl_Tree_Item::insert_above (const Fl_Tree_Prefs & *prefs*, const char ∗ *new_label*)

Insert a new item named `'new_label'` above this item.

Returns

the new item inserted, or 0 if an error occurred.

void Fl_Tree_Item::label (const char ∗ *name*)

Set the label to `'name'`.
Makes and manages an internal copy of `'name'`.

int Fl_Tree_Item::label_h ( ) const `[inline]`

The item's label height.

Version

1.3.3

int Fl_Tree_Item::label_w ( ) const `[inline]`

The item's maximum label width to right edge of Fl_Tree's inner width within scrollbars.

Version

1.3.3

int Fl_Tree_Item::label_x ( ) const `[inline]`

The item's label x position relative to the window.

Version

1.3.3

int Fl_Tree_Item::label_y ( ) const `[inline]`

The item's label y position relative to the window.

Version

1.3.3

void Fl_Tree_Item::labelbgcolor (Fl_Color *val*) `[inline]`

Set item's label background color.
A special case is made for color 0xffffffff which uses the parent tree's bg color.

Fl_Color Fl_Tree_Item::labelbgcolor ( ) const `[inline]`

Return item's label background text color.

If the color is 0xffffffff, the default behavior is the parent tree's bg color will be used. (An overloaded draw_item_content() can override this behavior.)

int Fl_Tree_Item::move (int *to,* int *from*)

Move the item 'from' to sibling position of 'to'.

Returns

- 0: Success
- -1: range error (e.g. if `'to'` or `'from'` out of range).
- (Other return values reserved for future use)

int Fl_Tree_Item::move (Fl_Tree_Item ∗ *item,* int *op* = *0,* int *pos* = *0*)

Move the current item above/below/into the specified 'item', where `'op'` determines the type of move:

- 0: move above `'item'` (`'pos'` ignored)
- 1: move below `'item'` (`'pos'` ignored)
- 2: move into `'item'` as a child (at optional position `'pos'`)

Returns

0 on success. a negative number on error:

- -1: one of the items has no parent
- -2: item's index could not be determined
- -3: bad 'op'
- -4: index range error
- -5: could not deparent
- -6: could not reparent at `'pos'`
- (Other return values reserved for future use.)

int Fl_Tree_Item::move_above (Fl_Tree_Item ∗ *item*)

Move the current item above the specified 'item'.

This is the equivalent of calling move(item,0,0).

Returns

0 on success.
On error returns a negative value; see move(Fl_Tree_Item∗,int,int) for possible error codes.

int Fl_Tree_Item::move_below (Fl_Tree_Item ∗ *item*)

Move the current item below the specified 'item'.

This is the equivalent of calling move(item,1,0).

Returns

0 on success.
On error returns a negative value; see move(Fl_Tree_Item∗,int,int) for possible error codes.

int Fl_Tree_Item::move_into (Fl_Tree_Item ∗ *item,* int *pos* = `0`)

Parent the current item as a child of the specified `'item'`.
This is the equivalent of calling move(item,2,pos).

Returns

0 on success.
On error returns a negative value; see move(Fl_Tree_Item∗,int,int) for possible error codes.

Fl_Tree_Item ∗ Fl_Tree_Item::next ( )

Return the next item in the tree.
This method can be used to walk the tree forward. For an example of how to use this method, see Fl_Tree::first().

Returns

the next item in the tree, or 0 if there's no more items.

Fl_Tree_Item ∗ Fl_Tree_Item::next_displayed (Fl_Tree_Prefs & *prefs*)

Same as next_visible().

Deprecated in 1.3.3 for confusing name, use next_visible() instead

Fl_Tree_Item ∗ Fl_Tree_Item::next_sibling ( )

Return this item's next sibling.
Moves to the next item below us at the same level (sibling). Use this to move down the tree without changing depth(). effectively skipping over this item's children/descendents.

Returns

item's next sibling, or 0 if none.

Fl_Tree_Item ∗ Fl_Tree_Item::next_visible (Fl_Tree_Prefs & *prefs*)

Return the next open(), visible() item.
(If this item has children and is closed, children are skipped)
This method can be used to walk the tree forward, skipping items that are not currently open/visible to the user.

Returns

the next open() visible() item below us, or 0 if there's no more items.

Version

1.3.3

void Fl_Tree_Item::parent (Fl_Tree_Item ∗ *val*) `[inline]`

Set the parent for this item.
Should only be used by Fl_Tree's internals.

const Fl_Tree_Prefs & Fl_Tree_Item::prefs ( ) const

Return the parent tree's prefs.

Returns

a reference to the parent tree's Fl_Tree_Prefs

Version

1.3.3 ABI feature

Fl_Tree_Item ∗ Fl_Tree_Item::prev ( )

Return the previous item in the tree.
This method can be used to walk the tree backwards. For an example of how to use this method, see Fl_Tree::last().

Returns

the previous item in the tree, or 0 if there's no item above this one (hit the root).

Fl_Tree_Item ∗ Fl_Tree_Item::prev_displayed (Fl_Tree_Prefs & *prefs*)

Same as prev_visible().

Deprecated in 1.3.3 for confusing name, use prev_visible()

Fl_Tree_Item ∗ Fl_Tree_Item::prev_sibling ( )

Return this item's previous sibling.
Moves to the previous item above us at the same level (sibling). Use this to move up the tree without changing depth().

Returns

This item's previous sibling, or 0 if none.

Fl_Tree_Item ∗ Fl_Tree_Item::prev_visible (Fl_Tree_Prefs & *prefs*)

Return the previous open(), visible() item.
(If this item above us has children and is closed, its children are skipped)
This method can be used to walk the tree backward, skipping items that are not currently open/visible to the user.

Returns

the previous open() visible() item above us, or 0 if there's no more items.

void Fl_Tree_Item::recalc_tree ( ) `[protected]`

Call this when our geometry is changed.
(Font size, label contents, etc) Schedules tree to recalculate itself, as changes to us may affect tree widget's scrollbar visibility and tab sizes.

Version

1.3.3 ABI

int Fl_Tree_Item::remove_child (Fl_Tree_Item ∗ *item*)

Remove `'item'` from the current item's children.

Returns

0 if removed, -1 if item not an immediate child.

int Fl_Tree_Item::remove_child (const char ∗ *name*)

Remove immediate child (and its children) by its label `'name'`.

If more than one item matches `'name'`, only the first matching item is removed.

Parameters

`in`	*name*	The label name of the immediate child to remove

Returns

0 if removed, -1 if not found.

Version

1.3.3

int Fl_Tree_Item::reparent (Fl_Tree_Item ∗ *newchild,* int *pos*)

Reparent specified item as a child of ourself at position `'pos'`.

Typically 'newchild' was recently orphaned with deparent().

Returns

- 0: on success
- -1: on error (e.g. if `'pos'` out of range) with no changes made.

Fl_Tree_Item ∗ Fl_Tree_Item::replace (Fl_Tree_Item ∗ *newitem*)

Replace the current item with a new item.

The current item is destroyed if successful. No checks are made to see if an item with the same name exists.

This method can be used to, for example, install 'custom' items into the tree derived from Fl_Tree_Item; see draw_item_content().

Parameters

`in`	*newitem*	The new item to replace the current item

Returns

newitem on success, NULL if could not be replaced.

See Also

Fl_Tree_Item::draw_item_content(), Fl_Tree::root(Fl_Tree_Item∗)

Version

1.3.3 ABI feature

Fl_Tree_Item ∗ Fl_Tree_Item::replace_child (Fl_Tree_Item ∗ *olditem*, Fl_Tree_Item ∗ *newitem*)

Replace existing child `'olditem'` with `'newitem'`.

The `'olditem'` is destroyed if successful. Can be used to put custom items (derived from Fl_Tree_Item) into the tree. No checks are made to see if an item with the same name exists.

Parameters

in	*olditem*	The item to be found and replaced
in	*newitem*	The new item to take the place of `'olditem'`

Returns

newitem on success and `'olditem'` is destroyed. NULL on error if `'olditem'` was not found as an immediate child.

See Also

replace(), Fl_Tree_Item::draw()

Version

1.3.3 ABI feature

void Fl_Tree_Item::select (int *val* = `1`) `[inline]`

Change the item's selection state to the optionally specified 'val'.
If 'val' is not specified, the item will be selected.

int Fl_Tree_Item::select_all ( ) `[inline]`

Select item and all its children.
Returns count of how many items were in the 'deselected' state, ie. how many items were "changed".

void Fl_Tree_Item::show_self (const char ∗ *indent* = `""`) const

Print the tree as 'ascii art' to stdout.
Used mainly for debugging.

void Fl_Tree_Item::show_widgets ( ) `[protected]`

Internal: Show the FLTK widget() for this item and all children.
Used by open() to re-show widgets that were hidden by a previous close()

void Fl_Tree_Item::swap_children (int *ax*, int *bx*)

Swap two of our children, given two child index values `'ax'` and `'bx'`.
Use e.g. for sorting.
This method is FAST, and does not involve lookups.
No range checking is done on either index value.

Parameters

in	*ax,bx*	the index of the items to swap

int Fl_Tree_Item::swap_children (Fl_Tree_Item ∗ *a*, Fl_Tree_Item ∗ *b*)

Swap two of our immediate children, given item pointers.
Use e.g. for sorting.
This method is SLOW because it involves linear lookups.
For speed, use swap_children(int,int) instead.

Parameters

in	*a,b*	The item ptrs of the two items to swap. Both must be immediate children of the current item.

Returns

- 0 : OK
- -1 : failed: item 'a' or 'b' is not our child.

const Fl_Tree* Fl_Tree_Item::tree ( ) const [inline]

Return the tree for this item.

Version

1.3.3 (ABI feature)

Fl_Tree* Fl_Tree_Item::tree ( ) [inline]

Return the tree for this item.

Version

1.3.4 (ABI feature)

void Fl_Tree_Item::update_prev_next (int *index*)

Update our _prev_sibling and _next_sibling pointers to point to neighbors given `index` as being our current position in the parent's item array.

Call this whenever items in the array are added/removed/moved/swapped/etc.

Parameters

in	*index*	Our index# in the parent. Special case if index=-1: become an orphan; null out all parent/sibling associations.

void Fl_Tree_Item::userdeicon (Fl_Image * *val*) [inline]

Set the usericon to draw when the item is deactivated.

Use '0' to disable. No internal copy is made; caller must manage icon's memory.

To create a typical 'grayed out' version of your usericon image, you can do the following:

```
// Create tree + usericon for items
Fl_Tree *tree = new Fl_Tree(..);
Fl_Image *usr_icon = new Fl_Pixmap(..); // your usericon
Fl_Image *de_icon  = usr_icon->copy();  // make a copy, and..
de_icon->inactive();                     // make it 'grayed out'
...
for ( .. ) {                 // item loop..
  item = tree->add("...");   // create new item
  item->usericon(usr_icon);  // assign usericon to items
  item->userdeicon(de_icon); // assign userdeicon to items
  ..
}
```

In the above example, the app should 'delete' the two icons when they're no longer needed (e.g. after the tree is destroyed)

Version

1.3.4

Fl_Image∗ Fl_Tree_Item::userdeicon ( ) const `[inline]`

Return the deactivated version of the user icon, if any.
Returns 0 if none.

void Fl_Tree_Item::usericon (Fl_Image ∗ *val*) `[inline]`

Set the item's user icon to an Fl_Image.
Use '0' to disable. No internal copy is made, caller must manage icon's memory.
Note, if you expect your items to be deactivated(), use userdeicon(Fl_Image∗) to set up a 'grayed out' version of your icon to be used for display.

See Also

userdeicon(Fl_Image∗)

int Fl_Tree_Item::visible_r ( ) const

See if item and all its parents are open() and visible().

Returns

1 – item and its parents are open() and visible() 0 – item (or one of its parents) are invisible or close()ed.

int Fl_Tree_Item::w ( ) const `[inline]`

The entire item's width to right edge of Fl_Tree's inner width within scrollbars.
The documentation for this class was generated from the following files:

- Fl_Tree_Item.H
- Fl_Tree_Item.cxx

31.145 Fl_Tree_Item_Array Class Reference

Manages an array of Fl_Tree_Item pointers.
`#include <Fl_Tree_Item_Array.H>`

Public Member Functions

- void add (Fl_Tree_Item ∗val)
  Add an item∗ to the end of the array.
- void clear ()
  Clear the entire array.
- int deparent (int pos)
  Deparent item at `'pos'` *from our list of children.*
- Fl_Tree_Item_Array (int new_chunksize=10)
  Constructor; creates an empty array.
- Fl_Tree_Item_Array (const Fl_Tree_Item_Array ∗o)
  Copy constructor. Makes new copy of array, with new instances of each item.
- void insert (int pos, Fl_Tree_Item ∗new_item)
  Insert an item at index position `pos`*.*
- void manage_item_destroy (int val)

Option to control if Fl_Tree_Item_Array's destructor will also destroy the Fl_Tree_Item's.

- int **manage_item_destroy** () const
- int move (int to, int from)

  Move item at 'from' to new position 'to' in the array.
- Fl_Tree_Item ∗ operator[ ] (int i)

  Return the item and index `i`.
- const Fl_Tree_Item ∗ operator[ ] (int i) const

  Const version of operator[ ](int i)
- void remove (int index)

  Remove the item at.
- int remove (Fl_Tree_Item ∗item)

  Remove the item from the array.
- int reparent (Fl_Tree_Item ∗item, Fl_Tree_Item ∗newparent, int pos)

  Reparent specified item as a child of ourself.
- void replace (int pos, Fl_Tree_Item ∗new_item)

  Replace the item at `index` with `newitem`.
- void swap (int ax, int bx)

  Swap the two items at index positions `ax` and `bx`.
- int total () const

  Return the total items in the array, or 0 if empty.
- ∼Fl_Tree_Item_Array ()

  Destructor. Calls each item's destructor, destroys internal _items array.

31.145.1 Detailed Description

Manages an array of Fl_Tree_Item pointers.

Because FLTK 1.x.x. has mandated that templates and STL not be used, we use this class to dynamically manage the arrays.

None of the methods do range checking on index values; the caller must be sure that index values are within the range 0<index<total() (unless otherwise noted).

31.145.2 Constructor & Destructor Documentation

Fl_Tree_Item_Array::Fl_Tree_Item_Array (int *new_chunksize* = `10`)

Constructor; creates an empty array.

The optional 'chunksize' can be specified to optimize memory allocation for potentially large arrays. Default chunksize is 10.

31.145.3 Member Function Documentation

void Fl_Tree_Item_Array::add (Fl_Tree_Item ∗ *val*)

Add an item∗ to the end of the array.

Assumes the item was created with 'new', and will remain allocated.. Fl_Tree_Item_Array will handle calling the item's destructor when the array is cleared or the item remove()'ed.

void Fl_Tree_Item_Array::clear ( )

Clear the entire array.

Each item will be deleted (destructors will be called), and the array will be cleared. total() will return 0.

int Fl_Tree_Item_Array::deparent (int *pos*)

Deparent item at `'pos'` from our list of children.

Similar to a remove() without the destruction of the item. This creates an orphaned item (still allocated, has no parent) which soon after is typically reparented elsewhere.

```
\returns 0 on success, -1 on error (e.g. if \p 'pos' out of range)
```

void Fl_Tree_Item_Array::insert (int *pos,* Fl_Tree_Item ∗ *new_item*)

Insert an item at index position `pos`.

Handles enlarging array if needed, total increased by 1. If `pos` == total(), an empty item is appended to the array.

void Fl_Tree_Item_Array::manage_item_destroy (int *val*) `[inline]`

Option to control if Fl_Tree_Item_Array's destructor will also destroy the Fl_Tree_Item's.

If set: items and item array is destroyed. If clear: only the item array is destroyed, not items themselves.

int Fl_Tree_Item_Array::move (int *to,* int *from*)

Move item at 'from' to new position 'to' in the array.

Due to how the moving an item shuffles the array around, a positional 'move' implies things that may not be obvious:

- When 'from' moved lower in tree, appears BELOW item that was at 'to'.
- When 'from' moved higher in tree, appears ABOVE item that was at 'to'.

  Returns

  0 on success, -1 on range error (e.g. if `'to'` or `'from'` out of range)

void Fl_Tree_Item_Array::remove (int *index*)

Remove the item at.

Parameters

`in`	*index*	from the array.

The item will be delete'd (if non-NULL), so its destructor will be called.

int Fl_Tree_Item_Array::remove (Fl_Tree_Item ∗ *item*)

Remove the item from the array.

Returns

0 if removed, or -1 if the item was not in the array.

int Fl_Tree_Item_Array::reparent (Fl_Tree_Item ∗ *item,* Fl_Tree_Item ∗ *newparent,* int *pos*)

Reparent specified item as a child of ourself.

Typically 'newchild' was recently orphaned with deparent().

```
\returns 0 on success, -1 on error (e.g. if \p 'pos' out of range)
```

void Fl_Tree_Item_Array::replace (int *index*, Fl_Tree_Item ∗ *newitem*)

Replace the item at `index` with `newitem`.

Old item at index position will be destroyed, and the new item will take it's place, and stitched into the linked list.

The documentation for this class was generated from the following files:

- Fl_Tree_Item_Array.H
- Fl_Tree_Item_Array.cxx

31.146 Fl_Tree_Prefs Class Reference

Tree widget's preferences.

`#include <Fl_Tree_Prefs.H>`

Public Member Functions

- Fl_Image ∗ closedeicon () const

  Return the deactivated version of the close icon, if any.
- Fl_Image ∗ closeicon () const

  Gets the default 'close' icon Returns the Fl_Image∗ of the icon, or 0 if none.
- void closeicon (Fl_Image ∗val)

  Sets the icon to be used as the 'close' icon.
- Fl_Color connectorcolor () const

  Get the connector color used for tree connection lines.
- void connectorcolor (Fl_Color val)

  Set the connector color used for tree connection lines.
- Fl_Tree_Connector connectorstyle () const

  Get the connector style.
- void connectorstyle (Fl_Tree_Connector val)

  Set the connector style.
- void connectorstyle (int val)

  Set the connector style [integer].
- int connectorwidth () const

  Get the tree connection line's width.
- void connectorwidth (int val)

  Set the tree connection line's width.
- void **do_item_draw_callback** (Fl_Tree_Item ∗o) const
- Fl_Tree_Prefs ()

  Fl_Tree_Prefs constructor.
- void **item_draw_callback** (Fl_Tree_Item_Draw_Callback ∗cb, void ∗data=0)
- Fl_Tree_Item_Draw_Callback ∗ **item_draw_callback** () const
- Fl_Tree_Item_Draw_Mode item_draw_mode () const

  Get the 'item draw mode' used for the tree.
- void item_draw_mode (Fl_Tree_Item_Draw_Mode val)

  Set the 'item draw mode' used for the tree to `val`.
- void ∗ **item_draw_user_data** () const
- Fl_Color item_labelbgcolor () const

  Get the default label background color.

- void item_labelbgcolor (Fl_Color val)

  Set the default label background color.
- Fl_Color item_labelfgcolor () const

  Get the default label foreground color.
- void item_labelfgcolor (Fl_Color val)

  Set the default label foreground color.
- Fl_Font item_labelfont () const

  Return the label's font.
- void item_labelfont (Fl_Font val)

  Set the label's font to `val`.
- Fl_Fontsize item_labelsize () const

  Return the label's size in pixels.
- void item_labelsize (Fl_Fontsize val)

  Set the label's size in pixels to `val`.
- Fl_Tree_Item_Reselect_Mode item_reselect_mode () const

  Returns the current item re/selection mode.
- void item_reselect_mode (Fl_Tree_Item_Reselect_Mode mode)

  Sets the item re/selection mode.
- Fl_Color labelbgcolor () const

  Obsolete: Get the default label background color. Please use item_labelbgcolor() instead.
- void labelbgcolor (Fl_Color val)

  Obsolete: Set the default label background color. Please use item_labelbgcolor(Fl_Color) instead.
- Fl_Color labelfgcolor () const

  Obsolete: Get the default label foreground color. Please use item_labelfgcolor() instead.
- void labelfgcolor (Fl_Color val)

  Obsolete: Set the default label foreground color. Please use item_labelfgcolor(Fl_Color) instead.
- Fl_Font labelfont () const

  Obsolete: Return the label's font. Please use item_labelfont() instead.
- void labelfont (Fl_Font val)

  Obsolete: Set the label's font to `val`*. Please use item_labelfont(Fl_Font) instead.*
- int labelmarginleft () const

  Get the label's left margin value in pixels.
- void labelmarginleft (int val)

  Set the label's left margin value in pixels.
- Fl_Fontsize labelsize () const

  Obsolete: Return the label's size in pixels. Please use item_labelsize() instead.
- void labelsize (Fl_Fontsize val)

  Obsolete: Set the label's size in pixels to `val`*. Please use item_labelsize(Fl_Fontsize) instead.*
- int linespacing () const

  Get the line spacing value in pixels.
- void linespacing (int val)

  Set the line spacing value in pixels.
- int marginbottom () const

  Get the bottom margin's value in pixels.
- void marginbottom (int val)

  Set the bottom margin's value in pixels This is the extra distance the vertical scroller lets you travel.

- int marginleft () const

  Get the left margin's value in pixels.
- void marginleft (int val)

  Set the left margin's value in pixels.
- int margintop () const

  Get the top margin's value in pixels.
- void margintop (int val)

  Set the top margin's value in pixels.
- int openchild_marginbottom () const

  Get the margin below an open child in pixels.
- void openchild_marginbottom (int val)

  Set the margin below an open child in pixels.
- Fl_Image ∗ opendeicon () const

  Return the deactivated version of the open icon, if any.
- Fl_Image ∗ openicon () const

  Get the current default 'open' icon.
- void openicon (Fl_Image ∗val)

  Sets the default icon to be used as the 'open' icon when items are add()ed to the tree.
- Fl_Boxtype selectbox () const

  Get the default selection box's box drawing style as an Fl_Boxtype.
- void selectbox (Fl_Boxtype val)

  Set the default selection box's box drawing style to `val`*.*
- Fl_Tree_Select selectmode () const

  Get the selection mode used for the tree.
- void selectmode (Fl_Tree_Select val)

  Set the selection mode used for the tree to `val`*.*
- char showcollapse () const

  Returns 1 if the collapse icon is enabled, 0 if not.
- void showcollapse (int val)

  Set if we should show the collapse icon or not.
- int showroot () const

  Returns 1 if the root item is to be shown, or 0 if not.
- void showroot (int val)

  Set if the root item should be shown or not.
- Fl_Tree_Sort sortorder () const

  Get the default sort order value.
- void sortorder (Fl_Tree_Sort val)

  Set the default sort order value.
- Fl_Image ∗ userdeicon () const

  Return the deactivated version of the user icon, if any.
- Fl_Image ∗ usericon () const

  Gets the default 'user icon' (default is 0)
- void usericon (Fl_Image ∗val)

  Sets the default 'user icon' Returns the Fl_Image∗ of the icon, or 0 if none (default).
- int usericonmarginleft () const

  Get the user icon's left margin value in pixels.

- void usericonmarginleft (int val)

  Set the user icon's left margin value in pixels.
- int widgetmarginleft () const

  Get the widget()'s left margin value in pixels.
- void widgetmarginleft (int val)

  Set the widget's left margin value in pixels.
- ~Fl_Tree_Prefs ()

  Fl_Tree_Prefs destructor.

31.146.1 Detailed Description

Tree widget's preferences.

Fl_Tree's Preferences class.

This class manages the Fl_Tree's defaults. You should probably be using the methods in Fl_Tree instead of trying to accessing tree's preferences settings directly.

31.146.2 Member Function Documentation

Fl_Image∗ Fl_Tree_Prefs::closedeicon ( ) const `[inline]`

Return the deactivated version of the close icon, if any.

Returns 0 if none.

void Fl_Tree_Prefs::closeicon (Fl_Image ∗ *val*)

Sets the icon to be used as the 'close' icon.

This overrides the built in default '[-]' icon.

Parameters

in	*val*	– The new image, or zero to use the default [-] icon.

void Fl_Tree_Prefs::item_draw_mode (Fl_Tree_Item_Draw_Mode *val*) `[inline]`

Set the 'item draw mode' used for the tree to `val`.

This affects how items in the tree are drawn, such as when a widget() is defined. See Fl_Tree_Item_-Draw_Mode for possible values.

Fl_Color Fl_Tree_Prefs::item_labelbgcolor (void) const `[inline]`

Get the default label background color.

This returns the Fl_Tree::color() unless item_labelbgcolor() has been set explicitly.

void Fl_Tree_Prefs::item_labelbgcolor (Fl_Color *val*) `[inline]`

Set the default label background color.

Once set, overrides the default behavior of using Fl_Tree::color().

int Fl_Tree_Prefs::marginbottom ( ) const `[inline]`

Get the bottom margin's value in pixels.

This is the extra distance the vertical scroller lets you travel.

void Fl_Tree_Prefs::marginbottom (int *val*) `[inline]`

Set the bottom margin's value in pixels This is the extra distance the vertical scroller lets you travel.

Fl_Image∗ Fl_Tree_Prefs::opendeicon ( ) const `[inline]`

Return the deactivated version of the open icon, if any.
Returns 0 if none.

Fl_Image∗ Fl_Tree_Prefs::openicon ( ) const `[inline]`

Get the current default 'open' icon.
Returns the Fl_Image∗ of the icon, or 0 if none.

void Fl_Tree_Prefs::openicon (Fl_Image ∗ *val*)

Sets the default icon to be used as the 'open' icon when items are add()ed to the tree.
This overrides the built in default '[+]' icon.

Parameters

`in`	*val*	– The new image, or zero to use the default [+] icon.

void Fl_Tree_Prefs::selectmode (Fl_Tree_Select *val*) `[inline]`

Set the selection mode used for the tree to `val`.
This affects how items in the tree are selected when clicked on and dragged over by the mouse. See Fl_Tree_Select for possible values.

void Fl_Tree_Prefs::showcollapse (int *val*) `[inline]`

Set if we should show the collapse icon or not.
If collapse icons are disabled, the user will not be able to interactively collapse items in the tree, unless the application provides some other means via open() and close().

Parameters

`in`	*val*	1: shows collapse icons (default), 0: hides collapse icons.

void Fl_Tree_Prefs::showroot (int *val*) `[inline]`

Set if the root item should be shown or not.

Parameters

`in`	*val*	1 – show the root item (default) 0 – hide the root item.

void Fl_Tree_Prefs::sortorder (Fl_Tree_Sort *val*) `[inline]`

Set the default sort order value.
Defines the order new items appear when add()ed to the tree. See Fl_Tree_Sort for possible values.

Fl_Image∗ Fl_Tree_Prefs::userdeicon ( ) const [inline]

Return the deactivated version of the user icon, if any.

Returns 0 if none.

The documentation for this class was generated from the following files:

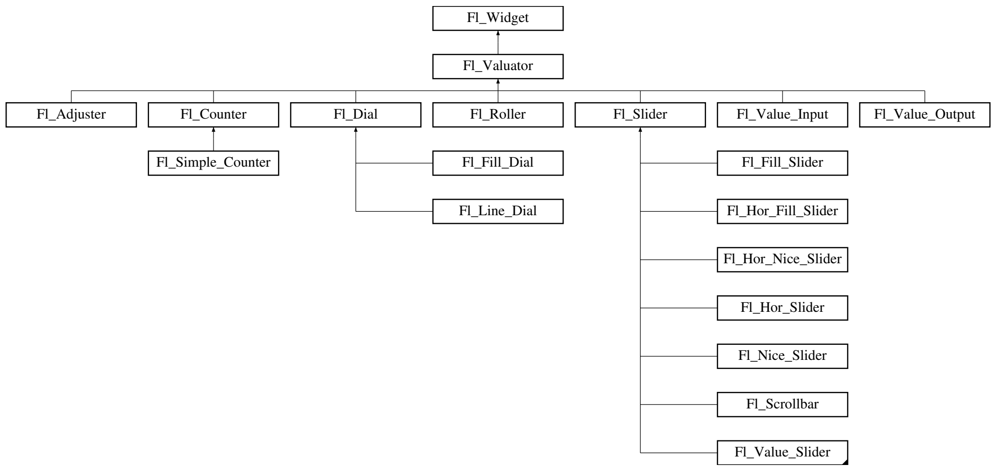
- Fl_Tree_Prefs.H
- Fl_Tree_Prefs.cxx

31.147 Fl_Valuator Class Reference

The Fl_Valuator class controls a single floating-point value and provides a consistent interface to set the value, range, and step, and insures that callbacks are done the same for every object.

`#include <Fl_Valuator.H>`

Inheritance diagram for Fl_Valuator:



Public Member Functions

- void bounds (double a, double b)

  Sets the minimum (a) and maximum (b) values for the valuator widget.
- double clamp (double)

  Clamps the passed value to the valuator range.
- virtual int format (char ∗)

  Uses internal rules to format the fields numerical value into the character array pointed to by the passed parameter.
- double increment (double, int)

  Adds n times the step value to the passed value.
- double maximum () const

  Gets the maximum value for the valuator.
- void maximum (double a)

  Sets the maximum value for the valuator.
- double minimum () const

  Gets the minimum value for the valuator.
- void minimum (double a)

  Sets the minimum value for the valuator.

- void precision (int digits)

  Sets the step value to 1.0 / 10^{digits}.
- void range (double a, double b)

  Sets the minimum and maximum values for the valuator.
- double round (double)

  Round the passed value to the nearest step increment.
- void step (int a)

  See double Fl_Valuator::step() const.
- void step (double a, int b)

  See double Fl_Valuator::step() const.
- void step (double s)

  See double Fl_Valuator::step() const.
- double step () const

  Gets or sets the step value.
- double value () const

  Gets the floating point(double) value.
- int value (double)

  Sets the current value.

Protected Member Functions

- Fl_Valuator (int X, int Y, int W, int H, const char ∗L)

  Creates a new Fl_Valuator widget using the given position, size, and label string.
- void handle_drag (double newvalue)

  Called during a drag operation, after an FL_WHEN_CHANGED event is received and before the callback.
- void handle_push ()

  Stores the current value in the previous value.
- void handle_release ()

  Called after an FL_WHEN_RELEASE event is received and before the callback.
- int horizontal () const

  Tells if the valuator is an FL_HORIZONTAL one.
- double previous_value () const

  Gets the previous floating point value before an event changed it.
- void set_value (double v)

  Sets the current floating point value.
- double softclamp (double)

  Clamps the value, but accepts v if the previous value is not already out of range.
- virtual void value_damage ()

  Asks for partial redraw.

Additional Inherited Members

31.147.1 Detailed Description

The Fl_Valuator class controls a single floating-point value and provides a consistent interface to set the value, range, and step, and insures that callbacks are done the same for every object.

There are probably more of these classes in FLTK than any others:

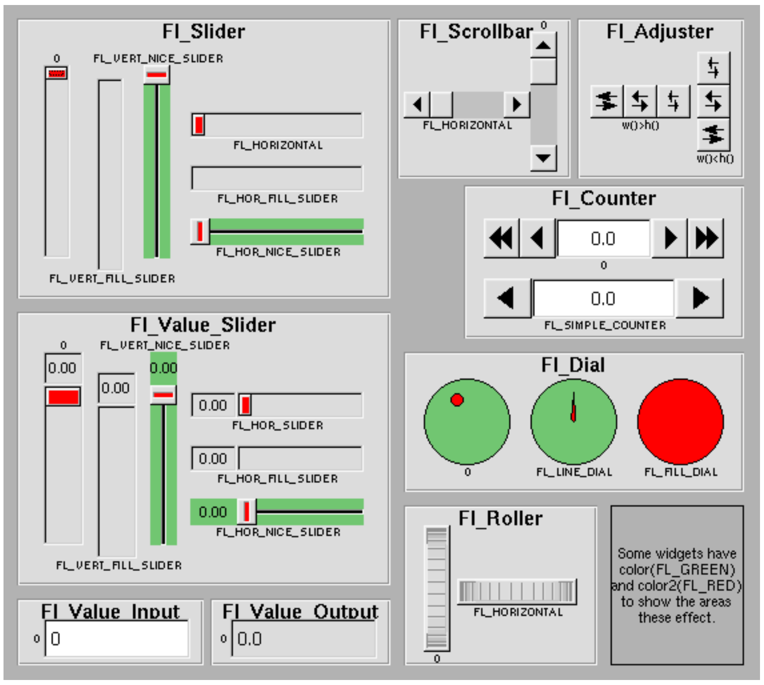


Figure 31.49: Valuators derived from Fl_Valuators

In the above diagram each box surrounds an actual subclass. These are further differentiated by setting the type() of the widget to the symbolic value labeling the widget. The ones labelled "0" are the default versions with a type(0). For consistency the symbol FL_VERTICAL is defined as zero.

31.147.2 Constructor & Destructor Documentation

Fl_Valuator::Fl_Valuator (int *X*, int *Y*, int *W*, int *H*, const char ∗ *L*) `[protected]`

Creates a new Fl_Valuator widget using the given position, size, and label string.

The default boxtype is FL_NO_BOX.

31.147.3 Member Function Documentation

void Fl_Valuator::bounds (double *a*, double *b*) `[inline]`

Sets the minimum (a) and maximum (b) values for the valuator widget.

double Fl_Valuator::clamp (double *v*)

Clamps the passed value to the valuator range.

int Fl_Valuator::format (char ∗ *buffer*) `[virtual]`

Uses internal rules to format the fields numerical value into the character array pointed to by the passed parameter.

The actual format used depends on the current step value. If the step value has been set to zero then a %g format is used. If the step value is non-zero, then a %.∗f format is used, where the precision is calculated to show sufficient digits for the current step value. An integer step value, such as 1 or 1.0, gives a precision of 0, so the formatted value will appear as an integer.

This method is used by the Fl_Valuator_... group of widgets to format the current value into a text string. The return value is the length of the formatted text. The formatted value is written into `buffer`. `buffer` should have space for at least 128 bytes.

You may override this function to create your own text formatting.

void Fl_Valuator::handle_drag (double *v*) `[protected]`

Called during a drag operation, after an FL_WHEN_CHANGED event is received and before the callback.

void Fl_Valuator::handle_release ( ) `[protected]`

Called after an FL_WHEN_RELEASE event is received and before the callback.

double Fl_Valuator::increment (double *v*, int *n*)

Adds n times the step value to the passed value.

If step was set to zero it uses fabs(maximum() - minimum()) / 100.

double Fl_Valuator::maximum ( ) const `[inline]`

Gets the maximum value for the valuator.

void Fl_Valuator::maximum (double *a*) `[inline]`

Sets the maximum value for the valuator.

double Fl_Valuator::minimum ( ) const `[inline]`

Gets the minimum value for the valuator.

void Fl_Valuator::minimum (double *a*) `[inline]`

Sets the minimum value for the valuator.

void Fl_Valuator::precision (int *digits*)

Sets the step value to 1.0 / 10^{digits} .

Precision `digits` is limited to 0...9 to avoid internal overflow errors. Values outside this range are clamped.

Note

For negative values of `digits` the step value is set to `A` = 1.0 and `B` = 1, i.e. 1.0/1 = 1.

void Fl_Valuator::range (double *a*, double *b*) `[inline]`

Sets the minimum and maximum values for the valuator.

When the user manipulates the widget, the value is limited to this range. This clamping is done *after* rounding to the step value (this makes a difference if the range is not a multiple of the step).

The minimum may be greater than the maximum. This has the effect of "reversing" the object so the larger values are in the opposite direction. This also switches which end of the filled sliders is filled.

Some widgets consider this a "soft" range. This means they will stop at the range, but if the user releases and grabs the control again and tries to move it further, it is allowed.

The range may affect the display. You must redraw() the widget after changing the range.

double Fl_Valuator::round (double *v*)

Round the passed value to the nearest step increment.

Does nothing if step is zero.

void Fl_Valuator::set_value (double *v*) `[inline]`, `[protected]`

Sets the current floating point value.

double Fl_Valuator::step ( ) const `[inline]`

Gets or sets the step value.

As the user moves the mouse the value is rounded to the nearest multiple of the step value. This is done *before* clamping it to the range. For most widgets the default step is zero.

For precision the step is stored as the ratio of a double `A` and an integer `B` = A/B. You can set these values directly. Currently setting a floating point value sets the nearest A/1 or 1/B value possible.

double Fl_Valuator::value ( ) const `[inline]`

Gets the floating point(double) value.

See int value(double)

int Fl_Valuator::value (double *v*)

Sets the current value.

The new value is *not* clamped or otherwise changed before storing it. Use clamp() or round() to modify the value before calling value(). The widget is redrawn if the new value is different than the current one. The initial value is zero.

changed() will return true if the user has moved the slider, but it will be turned off by value(x) and just before doing a callback (the callback can turn it back on if desired).

The documentation for this class was generated from the following files:

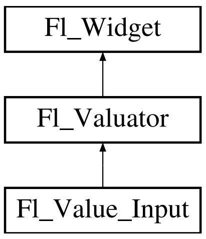
- Fl_Valuator.H
- Fl_Valuator.cxx

31.148 Fl_Value_Input Class Reference

The Fl_Value_Input widget displays a numeric value.

`#include <Fl_Value_Input.H>`

Inheritance diagram for Fl_Value_Input:



Public Member Functions

- Fl_Color cursor_color () const

    Gets the color of the text cursor.
- void cursor_color (Fl_Color n)

    Sets the color of the text cursor.
- Fl_Value_Input (int x, int y, int w, int h, const char ∗l=0)

    Creates a new Fl_Value_Input widget using the given position, size, and label string.
- int handle (int)

    Handles the specified event.
- void resize (int, int, int, int)

    Changes the size or position of the widget.
- int shortcut () const

    Returns the current shortcut key for the Input.
- void shortcut (int s)

    Sets the shortcut key to `s`.
- void soft (char s)

    See void Fl_Value_Input::soft(char s)
- char soft () const

    If "soft" is turned on, the user is allowed to drag the value outside the range.
- Fl_Color textcolor () const

    Gets the color of the text in the value box.
- void textcolor (Fl_Color n)

    Sets the color of the text in the value box.
- Fl_Font textfont () const

    Gets the typeface of the text in the value box.
- void textfont (Fl_Font s)

    Sets the typeface of the text in the value box.
- Fl_Fontsize textsize () const

    Gets the size of the text in the value box.
- void textsize (Fl_Fontsize s)

    Sets the size of the text in the value box.

Public Attributes

- Fl_Input **input**

Protected Member Functions

- void draw ()

    Draws the widget.

Additional Inherited Members

31.148.1 Detailed Description

The Fl_Value_Input widget displays a numeric value.

The user can click in the text field and edit it - there is in fact a hidden Fl_Input widget with type(FL_FLOAT_INPUT) or type(FL_INT_INPUT) in there - and when they hit return or tab the value updates to what they typed and the callback is done.

If step() is non-zero and integral, then the range of numbers is limited to integers instead of floating point numbers. As well as displaying the value as an integer, typed input is also limited to integer values, even if the hidden Fl_Input widget is of type(FL_FLOAT_INPUT).

If step() is non-zero, the user can also drag the mouse across the object and thus slide the value. The left button moves one step() per pixel, the middle by 10 step(), and the right button by 100 ∗ step(). It is therefore impossible to select text by dragging across it, although clicking can still move the insertion cursor.

If step() is non-zero and integral, then the range of numbers are limited to integers instead of floating point values.

Figure 31.50: Fl_Value_Input

31.148.2 Constructor & Destructor Documentation

Fl_Value_Input::Fl_Value_Input (int *X*, int *Y*, int *W*, int *H*, const char ∗ *l* = `0`)

Creates a new Fl_Value_Input widget using the given position, size, and label string.

The default boxtype is FL_DOWN_BOX.

31.148.3 Member Function Documentation

Fl_Color Fl_Value_Input::cursor_color ( ) const `[inline]`

Gets the color of the text cursor.

The text cursor is black by default.

void Fl_Value_Input::cursor_color (Fl_Color *n*) `[inline]`

Sets the color of the text cursor.

The text cursor is black by default.

void Fl_Value_Input::draw ( ) `[protected]`, `[virtual]`

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call redraw() instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw() method*, e.g. for an embedded scrollbar, you can do it (because draw() is virtual) like this:

```
Fl_Widget *s = &scroll;                 // scroll is an embedded Fl_Scrollbar
s->draw();                    // calls Fl_Scrollbar::draw()
```

Implements Fl_Widget.

int Fl_Value_Input::handle (int *event*) `[virtual]`

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited handle() method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters

`in`	*event*	the kind of event received

Return values

0	if the event was not used or understood
1	if the event was used and can be deleted

See Also

Fl_Event

Reimplemented from Fl_Widget.

void Fl_Value_Input::resize (int *x*, int *y*, int *w*, int *h*) `[virtual]`

Changes the size or position of the widget.

This is a virtual function so that the widget may implement its own handling of resizing. The default version does *not* call the redraw() method, but instead relies on the parent widget to do so because the parent may know a faster way to update the display, such as scrolling from the old position.

Some window managers under X11 call resize() a lot more often than needed. Please verify that the position or size of a widget did actually change before doing any extensive calculations.

position(X, Y) is a shortcut for resize(X, Y, w(), h()), and size(W, H) is a shortcut for resize(x(), y(), W, H).

Parameters

`in`	*x,y*	new position relative to the parent window
`in`	*w,h*	new size

See Also

position(int,int), size(int,int)

Reimplemented from Fl_Widget.

int Fl_Value_Input::shortcut ( ) const `[inline]`

Returns the current shortcut key for the Input.

See Also

Fl_Value_Input::shortcut(int)

void Fl_Value_Input::shortcut (int *s*) `[inline]`

Sets the shortcut key to `s`.

Setting this overrides the use of '&' in the label(). The value is a bitwise OR of a key and a set of shift flags, for example FL_ALT | 'a' , FL_ALT | (FL_F + 10), or just 'a'. A value of 0 disables the shortcut.

The key can be any value returned by Fl::event_key(), but will usually be an ASCII letter. Use a lower-case letter unless you require the shift key to be held down.

The shift flags can be any set of values accepted by Fl::event_state(). If the bit is on that shift key must be pushed. Meta, Alt, Ctrl, and Shift must be off if they are not in the shift flags (zero for the other bits indicates a "don't care" setting).

char Fl_Value_Input::soft ( ) const `[inline]`

If "soft" is turned on, the user is allowed to drag the value outside the range.

If they drag the value to one of the ends, let go, then grab again and continue to drag, they can get to any value. The default is true.

Fl_Color Fl_Value_Input::textcolor ( ) const `[inline]`

Gets the color of the text in the value box.

void Fl_Value_Input::textcolor (Fl_Color *n*) `[inline]`

Sets the color of the text in the value box.

Fl_Font Fl_Value_Input::textfont ( ) const `[inline]`

Gets the typeface of the text in the value box.

void Fl_Value_Input::textfont (Fl_Font *s*) `[inline]`

Sets the typeface of the text in the value box.

Fl_Fontsize Fl_Value_Input::textsize ( ) const `[inline]`

Gets the size of the text in the value box.

void Fl_Value_Input::textsize (Fl_Fontsize *s*) `[inline]`

Sets the size of the text in the value box.

The documentation for this class was generated from the following files:

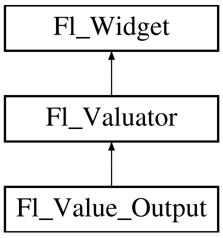
- Fl_Value_Input.H
- Fl_Value_Input.cxx

31.149 Fl_Value_Output Class Reference

The Fl_Value_Output widget displays a floating point value.

`#include <Fl_Value_Output.H>`

Inheritance diagram for Fl_Value_Output:



Public Member Functions

- Fl_Value_Output (int x, int y, int w, int h, const char ∗l=0)

  Creates a new Fl_Value_Output widget using the given position, size, and label string.
- int handle (int)

  Handles the specified event.
- void soft (uchar s)

  If "soft" is turned on, the user is allowed to drag the value outside the range.
- uchar soft () const

  If "soft" is turned on, the user is allowed to drag the value outside the range.
- Fl_Color textcolor () const

  Sets the color of the text in the value box.
- void textcolor (Fl_Color s)

  Gets the color of the text in the value box.
- Fl_Font textfont () const

  Gets the typeface of the text in the value box.
- void textfont (Fl_Font s)

  Sets the typeface of the text in the value box.
- Fl_Fontsize textsize () const

  Gets the size of the text in the value box.
- void **textsize** (Fl_Fontsize s)

Protected Member Functions

- void draw ()

  Draws the widget.

Additional Inherited Members

31.149.1 Detailed Description

The Fl_Value_Output widget displays a floating point value.

If step() is not zero, the user can adjust the value by dragging the mouse left and right. The left button moves one step() per pixel, the middle by 10 ∗ step(), and the right button by 100 ∗ step().

This is much lighter-weight than Fl_Value_Input because it contains no text editing code or character buffer.

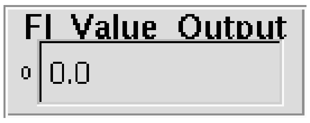


Figure 31.51: Fl_Value_Output

31.149.2 Constructor & Destructor Documentation

Fl_Value_Output::Fl_Value_Output (int *X*, int *Y*, int *W*, int *H*, const char ∗ *l* = 0)

Creates a new Fl_Value_Output widget using the given position, size, and label string.

The default boxtype is FL_NO_BOX.

Inherited destructor destroys the Valuator.

31.149.3 Member Function Documentation

void Fl_Value_Output::draw ( ) [protected], [virtual]

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call redraw() instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw() method*, e.g. for an embedded scrollbar, you can do it (because draw() is virtual) like this:

```
Fl_Widget *s = &scroll;                 // scroll is an embedded Fl_Scrollbar
s->draw();                          // calls Fl_Scrollbar::draw()
```

Implements Fl_Widget.

int Fl_Value_Output::handle (int *event*) [virtual]

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited handle() method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters

in	*event*	the kind of event received

Return values

0	if the event was not used or understood
1	if the event was used and can be deleted

See Also

Fl_Event

Reimplemented from Fl_Widget.

void Fl_Value_Output::soft (uchar *s*) [inline]

If "soft" is turned on, the user is allowed to drag the value outside the range.

If they drag the value to one of the ends, let go, then grab again and continue to drag, they can get to any value. Default is one.

uchar Fl_Value_Output::soft ( ) const [inline]

If "soft" is turned on, the user is allowed to drag the value outside the range.

If they drag the value to one of the ends, let go, then grab again and continue to drag, they can get to any value. Default is one.

Fl_Color Fl_Value_Output::textcolor ( ) const `[inline]`

Sets the color of the text in the value box.

void Fl_Value_Output::textcolor (Fl_Color *s*) `[inline]`

Gets the color of the text in the value box.

Fl_Font Fl_Value_Output::textfont ( ) const `[inline]`

Gets the typeface of the text in the value box.

void Fl_Value_Output::textfont (Fl_Font *s*) `[inline]`

Sets the typeface of the text in the value box.

Fl_Fontsize Fl_Value_Output::textsize ( ) const `[inline]`

Gets the size of the text in the value box.

The documentation for this class was generated from the following files:

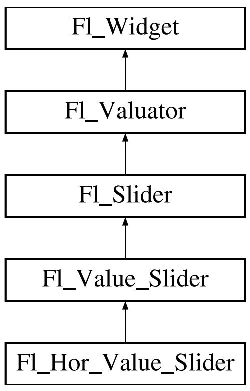
- Fl_Value_Output.H
- Fl_Value_Output.cxx

31.150 Fl_Value_Slider Class Reference

The Fl_Value_Slider widget is a Fl_Slider widget with a box displaying the current value.

`#include <Fl_Value_Slider.H>`

Inheritance diagram for Fl_Value_Slider:



Public Member Functions

- Fl_Value_Slider (int x, int y, int w, int h, const char *l=0)

  Creates a new Fl_Value_Slider widget using the given position, size, and label string.
- int handle (int)

  Handles the specified event.
- Fl_Color textcolor () const

  Gets the color of the text in the value box.
- void textcolor (Fl_Color s)

  Sets the color of the text in the value box.

- Fl_Font textfont () const

  Gets the typeface of the text in the value box.
- void textfont (Fl_Font s)

  Sets the typeface of the text in the value box.
- Fl_Fontsize textsize () const

  Gets the size of the text in the value box.
- void textsize (Fl_Fontsize s)

  Sets the size of the text in the value box.

Protected Member Functions

- void draw ()

  Draws the widget.

Additional Inherited Members

31.150.1 Detailed Description

The Fl_Value_Slider widget is a Fl_Slider widget with a box displaying the current value.

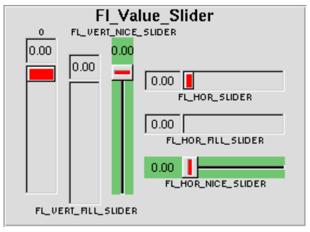


Figure 31.52: Fl_Value_Slider

31.150.2 Constructor & Destructor Documentation

Fl_Value_Slider::Fl_Value_Slider (int *X*, int *Y*, int *W*, int *H*, const char ∗ *l* = `0`)

Creates a new Fl_Value_Slider widget using the given position, size, and label string.

The default boxtype is FL_DOWN_BOX.

31.150.3 Member Function Documentation

void Fl_Value_Slider::draw ( ) `[protected]`, `[virtual]`

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call redraw() instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw() method*, e.g. for an embedded scrollbar, you can do it (because draw() is virtual) like this:

```
Fl_Widget *s = &scroll;                  // scroll is an embedded Fl_Scrollbar
s->draw();                     // calls Fl_Scrollbar::draw()
```

Reimplemented from Fl_Slider.

int Fl_Value_Slider::handle (int *event*) `[virtual]`

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited handle() method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters

`in`	*event*	the kind of event received

Return values

0	if the event was not used or understood
1	if the event was used and can be deleted

See Also

Fl_Event

Reimplemented from Fl_Slider.

Fl_Color Fl_Value_Slider::textcolor ( ) const `[inline]`

Gets the color of the text in the value box.

void Fl_Value_Slider::textcolor (Fl_Color *s*) `[inline]`

Sets the color of the text in the value box.

Fl_Font Fl_Value_Slider::textfont ( ) const `[inline]`

Gets the typeface of the text in the value box.

void Fl_Value_Slider::textfont (Fl_Font *s*) `[inline]`

Sets the typeface of the text in the value box.

Fl_Fontsize Fl_Value_Slider::textsize ( ) const `[inline]`

Gets the size of the text in the value box.

void Fl_Value_Slider::textsize (Fl_Fontsize *s*) `[inline]`

Sets the size of the text in the value box.

The documentation for this class was generated from the following files:

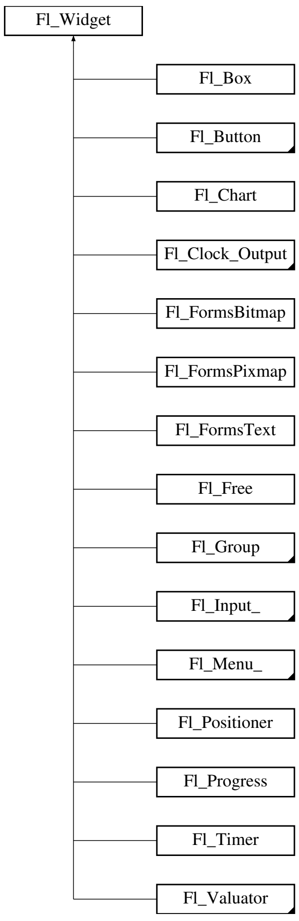
- Fl_Value_Slider.H
- Fl_Value_Slider.cxx

31.151 Fl_Widget Class Reference

Fl_Widget is the base class for all widgets in FLTK.

`#include <Fl_Widget.H>`

Inheritance diagram for Fl_Widget:



Public Member Functions

- void **_clear_fullscreen** ()
- void **_set_fullscreen** ()
- void activate ()

    Activates the widget.
- unsigned int active () const

    Returns whether the widget is active.
- int active_r () const

    Returns whether the widget and all of its parents are active.
- Fl_Align align () const

    Gets the label alignment.
- void align (Fl_Align alignment)

    Sets the label alignment.
- long argument () const

Gets the current user data (long) argument that is passed to the callback function.

- void argument (long v)

  Sets the current user data (long) argument that is passed to the callback function.
- virtual class Fl_Gl_Window ∗ as_gl_window ()

  Returns an Fl_Gl_Window pointer if this widget is an Fl_Gl_Window.
- virtual Fl_Group ∗ as_group ()

  Returns an Fl_Group pointer if this widget is an Fl_Group.
- virtual Fl_Window ∗ as_window ()

  Returns an Fl_Window pointer if this widget is an Fl_Window.
- Fl_Boxtype box () const

  Gets the box type of the widget.
- void box (Fl_Boxtype new_box)

  Sets the box type for the widget.
- Fl_Callback_p callback () const

  Gets the current callback function for the widget.
- void callback (Fl_Callback ∗cb, void ∗p)

  Sets the current callback function for the widget.
- void callback (Fl_Callback ∗cb)

  Sets the current callback function for the widget.
- void callback (Fl_Callback0 ∗cb)

  Sets the current callback function for the widget.
- void callback (Fl_Callback1 ∗cb, long p=0)

  Sets the current callback function for the widget.
- unsigned int changed () const

  Checks if the widget value changed since the last callback.
- void clear_active ()

  Marks the widget as inactive without sending events or changing focus.
- void clear_changed ()

  Marks the value of the widget as unchanged.
- void clear_damage (uchar c=0)

  Clears or sets the damage flags.
- void clear_output ()

  Sets a widget to accept input.
- void clear_visible ()

  Hides the widget.
- void clear_visible_focus ()

  Disables keyboard focus navigation with this widget.
- Fl_Color color () const

  Gets the background color of the widget.
- void color (Fl_Color bg)

  Sets the background color of the widget.
- void color (Fl_Color bg, Fl_Color sel)

  Sets the background and selection color of the widget.
- Fl_Color color2 () const

  For back compatibility only.
- void color2 (unsigned a)

For back compatibility only.

- int contains (const Fl_Widget ∗w) const

  Checks if w is a child of this widget.
- void copy_label (const char ∗new_label)

  Sets the current label.
- void copy_tooltip (const char ∗text)

  Sets the current tooltip text.
- uchar damage () const

  Returns non-zero if draw() needs to be called.
- void damage (uchar c)

  Sets the damage bits for the widget.
- void damage (uchar c, int x, int y, int w, int h)

  Sets the damage bits for an area inside the widget.
- int damage_resize (int, int, int, int)

  Internal use only.
- void deactivate ()

  Deactivates the widget.
- Fl_Image ∗ deimage ()

  Gets the image that is used as part of the widget label.
- const Fl_Image ∗ **deimage** () const
- void deimage (Fl_Image ∗img)

  Sets the image to use as part of the widget label.
- void deimage (Fl_Image &img)

  Sets the image to use as part of the widget label.
- void do_callback ()

  Calls the widget callback.
- void do_callback (Fl_Widget ∗o, long arg)

  Calls the widget callback.
- void do_callback (Fl_Widget ∗o, void ∗arg=0)

  Calls the widget callback.
- virtual void draw ()=0

  Draws the widget.
- void draw_label (int, int, int, int, Fl_Align) const

  Draws the label in an arbitrary bounding box with an arbitrary alignment.
- int h () const

  Gets the widget height.
- virtual int handle (int event)

  Handles the specified event.
- virtual void hide ()

  Makes a widget invisible.
- Fl_Image ∗ image ()

  Gets the image that is used as part of the widget label.
- const Fl_Image ∗ **image** () const
- void image (Fl_Image ∗img)

  Sets the image to use as part of the widget label.
- void image (Fl_Image &img)

  Sets the image to use as part of the widget label.

- int inside (const Fl_Widget ∗wgt) const

  Checks if this widget is a child of `wgt`*.*
- int is_label_copied () const

  Returns whether the current label was assigned with copy_label().
- const char ∗ label () const

  Gets the current label text.
- void label (const char ∗text)

  Sets the current label pointer.
- void label (Fl_Labeltype a, const char ∗b)

  Shortcut to set the label text and type in one call.
- Fl_Color labelcolor () const

  Gets the label color.
- void labelcolor (Fl_Color c)

  Sets the label color.
- Fl_Font labelfont () const

  Gets the font to use.
- void labelfont (Fl_Font f)

  Sets the font to use.
- Fl_Fontsize labelsize () const

  Gets the font size in pixels.
- void labelsize (Fl_Fontsize pix)

  Sets the font size in pixels.
- Fl_Labeltype labeltype () const

  Gets the label type.
- void labeltype (Fl_Labeltype a)

  Sets the label type.
- void measure_label (int &ww, int &hh) const

  Sets width ww and height hh accordingly with the label size.
- unsigned int output () const

  Returns if a widget is used for output only.
- Fl_Group ∗ parent () const

  Returns a pointer to the parent widget.
- void parent (Fl_Group ∗p)

  Internal use only - "for hacks only".
- void position (int X, int Y)

  Repositions the window or widget.
- void redraw ()

  Schedules the drawing of the widget.
- void redraw_label ()

  Schedules the drawing of the label.
- virtual void resize (int x, int y, int w, int h)

  Changes the size or position of the widget.
- Fl_Color selection_color () const

  Gets the selection color.
- void selection_color (Fl_Color a)

  Sets the selection color.

- void set_active ()

  Marks the widget as active without sending events or changing focus.
- void set_changed ()

  Marks the value of the widget as changed.
- void set_output ()

  Sets a widget to output only.
- void set_visible ()

  Makes the widget visible.
- void set_visible_focus ()

  Enables keyboard focus navigation with this widget.
- virtual void show ()

  Makes a widget visible.
- void size (int W, int H)

  Changes the size of the widget.
- int take_focus ()

  Gives the widget the keyboard focus.
- unsigned int takesevents () const

  Returns if the widget is able to take events.
- int test_shortcut ()

  Returns true if the widget's label contains the entered '&x' shortcut.
- const char ∗ tooltip () const

  Gets the current tooltip text.
- void tooltip (const char ∗text)

  Sets the current tooltip text.
- Fl_Window ∗ top_window () const

  Returns a pointer to the top-level window for the widget.
- Fl_Window ∗ top_window_offset (int &xoff, int &yoff) const

  Finds the x/y offset of the current widget relative to the top-level window.
- uchar type () const

  Gets the widget type.
- void type (uchar t)

  Sets the widget type.
- int use_accents_menu ()

  Returns non zero if MAC_USE_ACCENTS_MENU flag is set, 0 otherwise.
- void ∗ user_data () const

  Gets the user data for this widget.
- void user_data (void ∗v)

  Sets the user data for this widget.
- unsigned int visible () const

  Returns whether a widget is visible.
- void visible_focus (int v)

  Modifies keyboard focus navigation.
- unsigned int visible_focus ()

  Checks whether this widget has a visible focus.
- int visible_r () const

  Returns whether a widget and all its parents are visible.

- int w () const

  Gets the widget width.
- Fl_When when () const

  Returns the conditions under which the callback is called.
- void when (uchar i)

  Sets the flags used to decide when a callback is called.
- Fl_Window ∗ window () const

  Returns a pointer to the nearest parent window up the widget hierarchy.
- int x () const

  Gets the widget position in its window.
- int y () const

  Gets the widget position in its window.
- virtual ∼Fl_Widget ()

  Destroys the widget.

Static Public Member Functions

- static void default_callback (Fl_Widget ∗cb, void ∗d)

  The default callback for all widgets that don't set a callback.
- static unsigned int label_shortcut (const char ∗t)

  Returns the Unicode value of the '&x' shortcut in a given text.
- static int test_shortcut (const char ∗, const bool require_alt=false)

  Returns true if the given text `t` *contains the entered '&x' shortcut.*

Protected Types

- enum {
  INACTIVE = 1<<0, INVISIBLE = 1<<1, OUTPUT = 1<<2, NOBORDER = 1<<3,
  FORCE_POSITION = 1<<4, NON_MODAL = 1<<5, SHORTCUT_LABEL = 1<<6, CHANGE-
  D = 1<<7,
  OVERRIDE = 1<<8, VISIBLE_FOCUS = 1<<9, COPIED_LABEL = 1<<10, CLIP_CHILDREN
  = 1<<11,
  MENU_WINDOW = 1<<12, TOOLTIP_WINDOW = 1<<13, MODAL = 1<<14, NO_OVERL-
  AY = 1<<15,
  GROUP_RELATIVE = 1<<16, COPIED_TOOLTIP = 1<<17, FULLSCREEN = 1<<18, MAC_-
  USE_ACCENTS_MENU = 1<<19,
  USERFLAG3 = 1<<29, USERFLAG2 = 1<<30, USERFLAG1 = 1<<31 }

  flags possible values enumeration.

Protected Member Functions

- void clear_flag (unsigned int c)

  Clears a flag in the flags mask.
- void draw_backdrop () const

  If FL_ALIGN_IMAGE_BACKDROP is set, the image or deimage will be drawn.
- void draw_box () const

  Draws the widget box according its box style.
- void draw_box (Fl_Boxtype t, Fl_Color c) const

  Draws a box of type t, of color c at the widget's position and size.

- void draw_box (Fl_Boxtype t, int x, int y, int w, int h, Fl_Color c) const

  Draws a box of type t, of color c at the position X,Y and size W,H.
- void draw_focus ()

  draws a focus rectangle around the widget
- void draw_focus (Fl_Boxtype t, int x, int y, int w, int h) const

  Draws a focus box for the widget at the given position and size.
- void draw_label () const

  Draws the widget's label at the defined label position.
- void draw_label (int, int, int, int) const

  Draws the label in an arbitrary bounding box.
- Fl_Widget (int x, int y, int w, int h, const char ∗label=0L)

  Creates a widget at the given position and size.
- unsigned int flags () const

  Gets the widget flags mask.
- void h (int v)

  Internal use only.
- void set_flag (unsigned int c)

  Sets a flag in the flags mask.
- void w (int v)

  Internal use only.
- void x (int v)

  Internal use only.
- void y (int v)

  Internal use only.

Friends

- class **Fl_Group**

31.151.1 Detailed Description

Fl_Widget is the base class for all widgets in FLTK.

You can't create one of these because the constructor is not public. However you can subclass it.

All "property" accessing methods, such as color(), parent(), or argument() are implemented as trivial inline functions and thus are as fast and small as accessing fields in a structure. Unless otherwise noted, the property setting methods such as color(n) or label(s) are also trivial inline functions, even if they change the widget's appearance. It is up to the user code to call redraw() after these.

31.151.2 Member Enumeration Documentation

anonymous enum `[protected]`

flags possible values enumeration.

See activate(), output(), visible(), changed(), set_visible_focus()

Enumerator

INACTIVE the widget can't receive focus, and is disabled but potentially visible

INVISIBLE the widget is not drawn, but can receive a few special events

OUTPUT for output only

NOBORDER don't draw a decoration (Fl_Window)

FORCE_POSITION don't let the window manager position the window (Fl_Window)

NON_MODAL this is a hovering toolbar window (Fl_Window)

SHORTCUT_LABEL the label contains a shortcut we need to draw

CHANGED the widget value changed

OVERRIDE position window on top (Fl_Window)

VISIBLE_FOCUS accepts keyboard focus navigation if the widget can have the focus

COPIED_LABEL the widget label is internally copied, its destruction is handled by the widget

CLIP_CHILDREN all drawing within this widget will be clipped (Fl_Group)

MENU_WINDOW a temporary popup window, dismissed by clicking outside (Fl_Window)

TOOLTIP_WINDOW a temporary popup, transparent to events, and dismissed easily (Fl_Window)

MODAL a window blocking input to all other winows (Fl_Window)

NO_OVERLAY window not using a hardware overlay plane (Fl_Menu_Window)

GROUP_RELATIVE position this widget relative to the parent group, not to the window

COPIED_TOOLTIP the widget tooltip is internally copied, its destruction is handled by the widget

FULLSCREEN a fullscreen window (Fl_Window)

MAC_USE_ACCENTS_MENU On the Mac OS platform, pressing and holding a key on the keyboard opens an accented-character menu window (Fl_Input_, Fl_Text_Editor)

USERFLAG3 reserved for 3rd party extensions

USERFLAG2 reserved for 3rd party extensions

USERFLAG1 reserved for 3rd party extensions

31.151.3 Constructor & Destructor Documentation

Fl_Widget::Fl_Widget (int *x*, int *y*, int *w*, int *h*, const char ∗ *label* = `0L`) `[protected]`

Creates a widget at the given position and size.

The Fl_Widget is a protected constructor, but all derived widgets have a matching public constructor. It takes a value for x(), y(), w(), h(), and an optional value for label().

Parameters

`in`	*x,y*	the position of the widget relative to the enclosing window
`in`	*w,h*	size of the widget in pixels
`in`	*label*	optional text for the widget label

Fl_Widget::∼Fl_Widget ( ) `[virtual]`

Destroys the widget.

Destroys the widget, taking care of throwing focus before if any.

Destroying single widgets is not very common. You almost always want to destroy the parent group instead, which will destroy all of the child widgets and groups in that group.

Since

FLTK 1.3, the widget's destructor removes the widget from its parent group, if it is member of a group.

Destruction removes the widget from any parent group! And groups when destroyed destroy all their children. This is convenient and fast.

31.151.4 Member Function Documentation

void Fl_Widget::activate ( )

Activates the widget.
Changing this value will send FL_ACTIVATE to the widget if active_r() is true.

See Also

active(), active_r(), deactivate()

unsigned int Fl_Widget::active ( ) const `[inline]`

Returns whether the widget is active.
Return values

0	if the widget is inactive

See Also

active_r(), activate(), deactivate()

int Fl_Widget::active_r ( ) const

Returns whether the widget and all of its parents are active.
Return values

0	if this or any of the parent widgets are inactive

See Also

active(), activate(), deactivate()

Fl_Align Fl_Widget::align ( ) const `[inline]`

Gets the label alignment.

Returns

label alignment

See Also

label(), align(Fl_Align), Fl_Align

void Fl_Widget::align (Fl_Align *alignment*) `[inline]`

Sets the label alignment.
This controls how the label is displayed next to or inside the widget. The default value is FL_ALIGN_-CENTER, which centers the label inside the widget.

Parameters

in	*alignment*	new label alignment

See Also

align(), Fl_Align

long Fl_Widget::argument ( ) const `[inline]`

Gets the current user data (long) argument that is passed to the callback function.

Todo The user data value must be implemented using *intptr_t* or similar to avoid 64-bit machine incompatibilities.

void Fl_Widget::argument (long *v*) `[inline]`

Sets the current user data (long) argument that is passed to the callback function.

Todo The user data value must be implemented using *intptr_t* or similar to avoid 64-bit machine incompatibilities.

virtual class Fl_Gl_Window∗ Fl_Widget::as_gl_window ( ) `[inline], [virtual]`

Returns an Fl_Gl_Window pointer if this widget is an Fl_Gl_Window.

Use this method if you have a widget (pointer) and need to know whether this widget is derived from Fl_Gl_Window. If it returns non-NULL, then the widget in question is derived from Fl_Gl_Window.

Return values

NULL	if this widget is not derived from Fl_Gl_Window.

Note

This method is provided to avoid dynamic_cast.

See Also

Fl_Widget::as_group(), Fl_Widget::as_window()

Reimplemented in Fl_Gl_Window.

virtual Fl_Group∗ Fl_Widget::as_group ( ) `[inline], [virtual]`

Returns an Fl_Group pointer if this widget is an Fl_Group.

Use this method if you have a widget (pointer) and need to know whether this widget is derived from Fl_Group. If it returns non-NULL, then the widget in question is derived from Fl_Group, and you can use the returned pointer to access its children or other Fl_Group-specific methods.

Example:

```
void my_callback (Fl_Widget *w, void *) {
  Fl_Group *g = w->as_group();
  if (g)
    printf ("This group has %d children\n",g->children());
  else
    printf ("This widget is not a group!\n");
}
```

Return values

NULL	if this widget is not derived from Fl_Group.

Note

This method is provided to avoid dynamic_cast.

See Also

Fl_Widget::as_window(), Fl_Widget::as_gl_window()

Reimplemented in Fl_Group.

virtual Fl_Window∗ Fl_Widget::as_window ( ) `[inline], [virtual]`

Returns an Fl_Window pointer if this widget is an Fl_Window.

Use this method if you have a widget (pointer) and need to know whether this widget is derived from Fl_Window. If it returns non-NULL, then the widget in question is derived from Fl_Window, and you can use the returned pointer to access its children or other Fl_Window-specific methods.

Return values

NULL	if this widget is not derived from Fl_Window.

Note

This method is provided to avoid dynamic_cast.

See Also

Fl_Widget::as_group(), Fl_Widget::as_gl_window()

Reimplemented in Fl_Window.

Fl_Boxtype Fl_Widget::box ( ) const `[inline]`

Gets the box type of the widget.

Returns

the current box type

See Also

box(Fl_Boxtype), Fl_Boxtype

void Fl_Widget::box (Fl_Boxtype *new_box*) `[inline]`

Sets the box type for the widget.

This identifies a routine that draws the background of the widget. See Fl_Boxtype for the available types. The default depends on the widget, but is usually FL_NO_BOX or FL_UP_BOX.

Parameters

in	*new_box*	the new box type

See Also

box(), Fl_Boxtype

Fl_Callback_p Fl_Widget::callback ( ) const `[inline]`

Gets the current callback function for the widget.
Each widget has a single callback.

Returns

current callback

void Fl_Widget::callback (Fl_Callback ∗ *cb*, void ∗ *p*) `[inline]`

Sets the current callback function for the widget.
Each widget has a single callback.
Parameters

in	*cb*	new callback
in	*p*	user data

void Fl_Widget::callback (Fl_Callback ∗ *cb*) `[inline]`

Sets the current callback function for the widget.
Each widget has a single callback.
Parameters

in	*cb*	new callback

void Fl_Widget::callback (Fl_Callback0 ∗ *cb*) `[inline]`

Sets the current callback function for the widget.
Each widget has a single callback.
Parameters

in	*cb*	new callback

void Fl_Widget::callback (Fl_Callback1 ∗ *cb*, long *p* = *0*) `[inline]`

Sets the current callback function for the widget.
Each widget has a single callback.
Parameters

in	*cb*	new callback

in	*p*	user data

unsigned int Fl_Widget::changed ( ) const [inline]

Checks if the widget value changed since the last callback.

"Changed" is a flag that is turned on when the user changes the value stored in the widget. This is only used by subclasses of Fl_Widget that store values, but is in the base class so it is easier to scan all the widgets in a panel and do_callback() on the changed ones in response to an "OK" button.

Most widgets turn this flag off when they do the callback, and when the program sets the stored value.

Return values

0	if the value did not change

See Also

set_changed(), clear_changed()

void Fl_Widget::clear_active ( ) [inline]

Marks the widget as inactive without sending events or changing focus.

This is mainly for specialized use, for normal cases you want deactivate().

See Also

deactivate()

void Fl_Widget::clear_changed ( ) [inline]

Marks the value of the widget as unchanged.

See Also

changed(), set_changed()

void Fl_Widget::clear_damage (uchar *c* = *0*) [inline]

Clears or sets the damage flags.

Damage flags are cleared when parts of the widget drawing is repaired.

The optional argument `c` specifies the bits that **are set** after the call (default: 0) and **not** the bits that are cleared!

Note

Therefore it is possible to set damage bits with this method, but this should be avoided. Use damage(uchar) instead.

Parameters

in	*c*	new bitmask of damage flags (default: 0)

See Also

damage(uchar), damage()

void Fl_Widget::clear_output ( ) `[inline]`

Sets a widget to accept input.

See Also

set_output(), output()

void Fl_Widget::clear_visible ( ) `[inline]`

Hides the widget.

You must still redraw the parent to see a change in the window. Normally you want to use the hide() method instead.

void Fl_Widget::clear_visible_focus ( ) `[inline]`

Disables keyboard focus navigation with this widget.

Normally, all widgets participate in keyboard focus navigation.

See Also

set_visible_focus(), visible_focus(), visible_focus(int)

Fl_Color Fl_Widget::color ( ) const `[inline]`

Gets the background color of the widget.

Returns

current background color

See Also

color(Fl_Color), color(Fl_Color, Fl_Color)

void Fl_Widget::color (Fl_Color *bg*) `[inline]`

Sets the background color of the widget.

The color is passed to the box routine. The color is either an index into an internal table of RGB colors or an RGB color value generated using fl_rgb_color().

The default for most widgets is FL_BACKGROUND_COLOR. Use Fl::set_color() to redefine colors in the color map.

Parameters

in	*bg*	background color

See Also

color(), color(Fl_Color, Fl_Color), selection_color(Fl_Color)

void Fl_Widget::color (Fl_Color *bg*, Fl_Color *sel*) `[inline]`

Sets the background and selection color of the widget.

The two color form sets both the background and selection colors.

Parameters

in	*bg*	background color
in	*sel*	selection color

See Also

color(unsigned), selection_color(unsigned)

Fl_Color Fl_Widget::color2 ( ) const [inline]

For back compatibility only.

Deprecated Use selection_color() instead.

void Fl_Widget::color2 (unsigned *a*) [inline]

For back compatibility only.

Deprecated Use selection_color(unsigned) instead.

int Fl_Widget::contains (const Fl_Widget * *w*) const

Checks if w is a child of this widget.
Parameters

in	*w*	potential child widget

Returns

Returns 1 if `w` is a child of this widget, or is equal to this widget. Returns 0 if `w` is NULL.

void Fl_Widget::copy_label (const char * *new_label*)

Sets the current label.
Unlike label(), this method allocates a copy of the label string instead of using the original string pointer.
The internal copy will automatically be freed whenever you assign a new label or when the widget is destroyed.
Parameters

in	*new_label*	the new label text

See Also

label()

void Fl_Widget::copy_tooltip (const char * *text*)

Sets the current tooltip text.
Unlike tooltip(), this method allocates a copy of the tooltip string instead of using the original string pointer.
The internal copy will automatically be freed whenever you assign a new tooltip or when the widget is destroyed.
If no tooltip is set, the tooltip of the parent is inherited. Setting a tooltip for a group and setting no tooltip for a child will show the group's tooltip instead. To avoid this behavior, you can set the child's tooltip to an empty string ("").

Parameters

in	*text*	New tooltip text (an internal copy is made and managed)

See Also

tooltip(const char∗), tooltip()

uchar Fl_Widget::damage ( ) const `[inline]`

Returns non-zero if draw() needs to be called.

The damage value is actually a bit field that the widget subclass can use to figure out what parts to draw.

Returns

a bitmap of flags describing the kind of damage to the widget

See Also

damage(uchar), clear_damage(uchar)

void Fl_Widget::damage (uchar *c*)

Sets the damage bits for the widget.

Setting damage bits will schedule the widget for the next redraw.

Parameters

in	*c*	bitmask of flags to set

See Also

damage(), clear_damage(uchar)

void Fl_Widget::damage (uchar *c*, int *x*, int *y*, int *w*, int *h*)

Sets the damage bits for an area inside the widget.

Setting damage bits will schedule the widget for the next redraw.

Parameters

in	*c*	bitmask of flags to set
in	*x,y,w,h*	size of damaged area

See Also

damage(), clear_damage(uchar)

int Fl_Widget::damage_resize (int *X*, int *Y*, int *W*, int *H*)

Internal use only.

void Fl_Widget::deactivate ( )

Deactivates the widget.

Inactive widgets will be drawn "grayed out", e.g. with less contrast than the active widget. Inactive widgets will not receive any keyboard or mouse button events. Other events (including FL_ENTER, FL_MOVE, FL_LEAVE, FL_SHORTCUT, and others) will still be sent. A widget is only active if active() is true on it *and all of its parents*.

Changing this value will send FL_DEACTIVATE to the widget if active_r() is true.

Currently you cannot deactivate Fl_Window widgets.

See Also

activate(), active(), active_r()

void Fl_Widget::default_callback (Fl_Widget ∗ *cb*, void ∗ *d*) `[static]`

The default callback for all widgets that don't set a callback.

This callback function puts a pointer to the widget on the queue returned by Fl::readqueue().

Relying on the default callback and reading the callback queue with Fl::readqueue() is not recommended. If you need a callback, you should set one with Fl_Widget::callback(Fl_Callback ∗cb, void ∗data) or one of its variants.

Parameters

`in`	*cb*	the widget given to the callback
`in`	*d*	user data associated with that callback

See Also

callback(), do_callback(), Fl::readqueue()

Fl_Image∗ Fl_Widget::deimage ( ) `[inline]`

Gets the image that is used as part of the widget label.

This image is used when drawing the widget in the inactive state.

Returns

the current image for the deactivated widget

void Fl_Widget::deimage (Fl_Image ∗ *img*) `[inline]`

Sets the image to use as part of the widget label.

This image is used when drawing the widget in the inactive state.

Parameters

`in`	*img*	the new image for the deactivated widget

void Fl_Widget::deimage (Fl_Image & *img*) `[inline]`

Sets the image to use as part of the widget label.

This image is used when drawing the widget in the inactive state.

Parameters

in	*img*	the new image for the deactivated widget

void Fl_Widget::do_callback ( ) `[inline]`

Calls the widget callback.

Causes a widget to invoke its callback function with default arguments.

See Also

callback()

void Fl_Widget::do_callback (Fl_Widget ∗ *o*, long *arg*) `[inline]`

Calls the widget callback.

Causes a widget to invoke its callback function with arbitrary arguments.

Parameters

in	*o*	call the callback with `o` as the widget argument
in	*arg*	call the callback with `arg` as the user data argument

See Also

callback()

void Fl_Widget::do_callback (Fl_Widget ∗ *o*, void ∗ *arg* = `0`)

Calls the widget callback.

Causes a widget to invoke its callback function with arbitrary arguments.

Parameters

in	*o*	call the callback with `o` as the widget argument
in	*arg*	use `arg` as the user data argument

See Also

callback()

virtual void Fl_Widget::draw ( ) `[pure virtual]`

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call redraw() instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw() method*, e.g. for an embedded scrollbar, you can do it (because draw() is virtual) like this:

```
Fl_Widget *s = &scroll;              // scroll is an embedded Fl_Scrollbar
s->draw();                    // calls Fl_Scrollbar::draw()
```

Implemented in Fl_Table, Fl_FormsText, Fl_Tree, Fl_Text_Display, Fl_Gl_Window, Fl_Help_View, Fl_Input, Fl_Tabs, Fl_Browser_, Fl_Window, Fl_Scroll, Fl_Button, Fl_Choice, Fl_Chart, Fl_Slider, Fl_Menu_Bar, Fl_Value_Input, Fl_File_Input, Fl_Counter, Fl_Free, Fl_Menu_Button, Fl_Clock_Output, Fl_Group, Fl_Dial, Fl_Cairo_Window, Fl_Sys_Menu_Bar, Fl_Pack, Fl_Scrollbar, Fl_Positioner, Fl_Adjuster, Fl_Timer, Fl_Value_Output, Fl_Glut_Window, Fl_Progress, Fl_Light_Button, Fl_Value_Slider, Fl_Roller, Fl_Box, Fl_Return_Button, Fl_FormsPixmap, and Fl_FormsBitmap.

void Fl_Widget::draw_box (Fl_Boxtype *t*, Fl_Color *c*) const `[protected]`

Draws a box of type t, of color c at the widget's position and size.

void Fl_Widget::draw_box (Fl_Boxtype *t*, int *X*, int *Y*, int *W*, int *H*, Fl_Color *c*) const `[protected]`

Draws a box of type t, of color c at the position X,Y and size W,H.

void Fl_Widget::draw_label ( ) const `[protected]`

Draws the widget's label at the defined label position.
This is the normal call for a widget's draw() method.

void Fl_Widget::draw_label (int *X*, int *Y*, int *W*, int *H*) const `[protected]`

Draws the label in an arbitrary bounding box.
draw() can use this instead of draw_label(void) to change the bounding box

void Fl_Widget::draw_label (int *X*, int *Y*, int *W*, int *H*, Fl_Align *a*) const

Draws the label in an arbitrary bounding box with an arbitrary alignment.
Anybody can call this to force the label to draw anywhere.

void Fl_Widget::h (int *v*) `[inline], [protected]`

Internal use only.
Use position(int,int), size(int,int) or resize(int,int,int,int) instead.

int Fl_Widget::h ( ) const `[inline]`

Gets the widget height.

Returns

the height of the widget in pixels.

int Fl_Widget::handle (int *event*) `[virtual]`

Handles the specified event.
You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.
When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.
Most of the time, you want to call the inherited handle() method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.
Parameters

`in`	*event*	the kind of event received

Return values

0	if the event was not used or understood
1	if the event was used and can be deleted

See Also

Fl_Event

Reimplemented in Fl_Tree, Fl_Table, Fl_Help_View, Fl_Input, Fl_Window, Fl_Tabs, Fl_Browser_, Fl_Text_Display, Fl_Scroll, Fl_Spinner, Fl_Table_Row, Fl_Clock, Fl_Check_Browser, Fl_Button, Fl_Choice, Fl_Gl_Window, Fl_Slider, Fl_Menu_Button, Fl_Menu_Bar, Fl_Group, Fl_Value_Input, Fl_Counter, Fl_File_Input, Fl_Free, Fl_Dial, Fl_Scrollbar, Fl_Text_Editor, Fl_Positioner, Fl_Box, Fl_Value_Output, Fl_Timer, Fl_Glut_Window, Fl_Adjuster, Fl_Secret_Input, Fl_Light_Button, Fl_Value_Slider, Fl_Roller, Fl_Return_Button, Fl_Repeat_Button, and Fl_Tile.

void Fl_Widget::hide ( ) `[virtual]`

Makes a widget invisible.

See Also

show(), visible(), visible_r()

Reimplemented in Fl_Window, Fl_Browser, Fl_Gl_Window, Fl_Overlay_Window, Fl_Double_Window, and Fl_Menu_Window.

Fl_Image∗ Fl_Widget::image ( ) `[inline]`

Gets the image that is used as part of the widget label.
This image is used when drawing the widget in the active state.

Returns

the current image

void Fl_Widget::image (Fl_Image ∗ *img*) `[inline]`

Sets the image to use as part of the widget label.
This image is used when drawing the widget in the active state.
Parameters

`in`	*img*	the new image for the label

void Fl_Widget::image (Fl_Image & *img*) `[inline]`

Sets the image to use as part of the widget label.
This image is used when drawing the widget in the active state.
Parameters

`in`	*img*	the new image for the label

int Fl_Widget::inside (const Fl_Widget ∗ *wgt*) const `[inline]`

Checks if this widget is a child of `wgt`.
Returns 1 if this widget is a child of `wgt`, or is equal to `wgt`. Returns 0 if `wgt` is NULL.

Parameters

`in`	*wgt*	the possible parent widget.

See Also

contains()

int Fl_Widget::is_label_copied ( ) const `[inline]`

Returns whether the current label was assigned with copy_label().

This can be useful for temporarily overwriting the widget's label and restoring it later.

Return values

0	current label was assigned with label().
1	current label was assigned with copy_label().

const char∗ Fl_Widget::label ( ) const `[inline]`

Gets the current label text.

Returns

a pointer to the current label text

See Also

label(const char ∗), copy_label(const char ∗)

void Fl_Widget::label (const char ∗ *text*)

Sets the current label pointer.

The label is shown somewhere on or next to the widget. The passed pointer is stored unchanged in the widget (the string is *not* copied), so if you need to set the label to a formatted value, make sure the buffer is static, global, or allocated. The copy_label() method can be used to make a copy of the label string automatically.

Parameters

`in`	*text*	pointer to new label text

See Also

copy_label()

void Fl_Widget::label (Fl_Labeltype *a*, const char ∗ *b*) `[inline]`

Shortcut to set the label text and type in one call.

See Also

label(const char ∗), labeltype(Fl_Labeltype)

unsigned int Fl_Widget::label_shortcut (const char ∗ *t*) `[static]`

Returns the Unicode value of the '&x' shortcut in a given text.

The given text `t` (usually a widget's label or a menu text) is searched for a '&x' shortcut label, and if found, the Unicode value (code point) of the '&x' shortcut is returned.

Parameters

t	text or label to search for '&x' shortcut.

Returns

Unicode (UCS-4) value of shortcut in `t` or 0.

Note

Internal use only.

Fl_Color Fl_Widget::labelcolor ( ) const `[inline]`

Gets the label color.

The default color is FL_FOREGROUND_COLOR.

Returns

the current label color

void Fl_Widget::labelcolor (Fl_Color *c*) `[inline]`

Sets the label color.

The default color is FL_FOREGROUND_COLOR.

Parameters

`in`	*c*	the new label color

Fl_Font Fl_Widget::labelfont ( ) const `[inline]`

Gets the font to use.

Fonts are identified by indexes into a table. The default value uses a Helvetica typeface (Arial for Microsoft® Windows®). The function Fl::set_font() can define new typefaces.

Returns

current font used by the label

See Also

Fl_Font

void Fl_Widget::labelfont (Fl_Font *f*) `[inline]`

Sets the font to use.

Fonts are identified by indexes into a table. The default value uses a Helvetica typeface (Arial for Microsoft® Windows®). The function Fl::set_font() can define new typefaces.

Parameters

in	*f*	the new font for the label

See Also

Fl_Font

Fl_Fontsize Fl_Widget::labelsize ( ) const `[inline]`

Gets the font size in pixels.
The default size is 14 pixels.

Returns

the current font size

void Fl_Widget::labelsize (Fl_Fontsize *pix*) `[inline]`

Sets the font size in pixels.
Parameters

in	*pix*	the new font size

See Also

Fl_Fontsize labelsize()

Fl_Labeltype Fl_Widget::labeltype ( ) const `[inline]`

Gets the label type.

Returns

the current label type.

See Also

Fl_Labeltype

void Fl_Widget::labeltype (Fl_Labeltype *a*) `[inline]`

Sets the label type.
The label type identifies the function that draws the label of the widget. This is generally used for special effects such as embossing or for using the label() pointer as another form of data such as an icon. The value FL_NORMAL_LABEL prints the label as plain text.
Parameters

in	*a*	new label type

See Also

Fl_Labeltype

void Fl_Widget::measure_label (int & *ww*, int & *hh*) const `[inline]`

Sets width ww and height hh accordingly with the label size.

Labels with images will return w() and h() of the image.

This calls fl_measure() internally. For more information about the arguments `ww` and `hh` and word wrapping

See Also

fl_measure(const char∗, int&, int&, int)

unsigned int Fl_Widget::output ( ) const `[inline]`

Returns if a widget is used for output only.

output() means the same as !active() except it does not change how the widget is drawn. The widget will not receive any events. This is useful for making scrollbars or buttons that work as displays rather than input devices.

Return values

0	if the widget is used for input and output

See Also

set_output(), clear_output()

Fl_Group∗ Fl_Widget::parent ( ) const `[inline]`

Returns a pointer to the parent widget.

Usually this is a Fl_Group or Fl_Window.

Return values

NULL	if the widget has no parent

See Also

Fl_Group::add(Fl_Widget∗)

void Fl_Widget::parent (Fl_Group ∗ *p*) `[inline]`

Internal use only - "for hacks only".

It is ***STRONGLY*** *recommended* not to use this method, because it short-circuits Fl_Group's normal widget adding and removing methods, if the widget is already a child widget of another Fl_Group.

Use Fl_Group::add(Fl_Widget∗) and/or Fl_Group::remove(Fl_Widget∗) instead.

void Fl_Widget::position (int *X*, int *Y*) `[inline]`

Repositions the window or widget.

position(X, Y) is a shortcut for resize(X, Y, w(), h()).

Parameters

`in`	*X,Y*	new position relative to the parent window

See Also

resize(int,int,int,int), size(int,int)

void Fl_Widget::redraw ( )

Schedules the drawing of the widget.

Marks the widget as needing its draw() routine called.

void Fl_Widget::redraw_label ( )

Schedules the drawing of the label.

Marks the widget or the parent as needing a redraw for the label area of a widget.

void Fl_Widget::resize (int *x*, int *y*, int *w*, int *h*) `[virtual]`

Changes the size or position of the widget.

This is a virtual function so that the widget may implement its own handling of resizing. The default version does *not* call the redraw() method, but instead relies on the parent widget to do so because the parent may know a faster way to update the display, such as scrolling from the old position.

Some window managers under X11 call resize() a lot more often than needed. Please verify that the position or size of a widget did actually change before doing any extensive calculations.

position(X, Y) is a shortcut for resize(X, Y, w(), h()), and size(W, H) is a shortcut for resize(x(), y(), W, H).

Parameters

`in`	*x,y*	new position relative to the parent window
`in`	*w,h*	new size

See Also

position(int,int), size(int,int)

Reimplemented in Fl_Table, Fl_Tree, Fl_Help_View, Fl_Text_Display, Fl_Window, Fl_Browser_, Fl_Input_Choice, Fl_Input_, Fl_Spinner, Fl_Scroll, Fl_Group, Fl_Gl_Window, Fl_Value_Input, Fl_Overlay_Window, Fl_Double_Window, and Fl_Tile.

Fl_Color Fl_Widget::selection_color ( ) const `[inline]`

Gets the selection color.

Returns

the current selection color

See Also

selection_color(Fl_Color), color(Fl_Color, Fl_Color)

void Fl_Widget::selection_color (Fl_Color *a*) `[inline]`

Sets the selection color.

The selection color is defined for Forms compatibility and is usually used to color the widget when it is selected, although some widgets use this color for other purposes. You can set both colors at once with color(Fl_Color bg, Fl_Color sel).

Parameters

in	*a*	the new selection color

See Also

selection_color(), color(Fl_Color, Fl_Color)

void Fl_Widget::set_active ( ) `[inline]`

Marks the widget as active without sending events or changing focus.
This is mainly for specialized use, for normal cases you want activate().

See Also

activate()

void Fl_Widget::set_changed ( ) `[inline]`

Marks the value of the widget as changed.

See Also

changed(), clear_changed()

void Fl_Widget::set_output ( ) `[inline]`

Sets a widget to output only.

See Also

output(), clear_output()

void Fl_Widget::set_visible ( ) `[inline]`

Makes the widget visible.
You must still redraw the parent widget to see a change in the window. Normally you want to use the show() method instead.

void Fl_Widget::set_visible_focus ( ) `[inline]`

Enables keyboard focus navigation with this widget.
Note, however, that this will not necessarily mean that the widget will accept focus, but for widgets that can accept focus, this method enables it if it has been disabled.

See Also

visible_focus(), clear_visible_focus(), visible_focus(int)

void Fl_Widget::show ( ) `[virtual]`

Makes a widget visible.

An invisible widget never gets redrawn and does not get keyboard or mouse events, but can receive a few other events like FL_SHOW.

The visible() method returns true if the widget is set to be visible. The visible_r() method returns true if the widget and all of its parents are visible. A widget is only visible if visible() is true on it *and all of its parents*.

Changing it will send FL_SHOW or FL_HIDE events to the widget. *Do not change it if the parent is not visible, as this will send false FL_SHOW or FL_HIDE events to the widget*. redraw() is called if necessary on this or the parent.

See Also

hide(), visible(), visible_r()

Reimplemented in Fl_Window, Fl_Browser, Fl_Gl_Window, Fl_Overlay_Window, Fl_Double_Window, Fl_Single_Window, and Fl_Menu_Window.

void Fl_Widget::size (int *W*, int *H*) `[inline]`

Changes the size of the widget.

size(W, H) is a shortcut for resize(x(), y(), W, H).

Parameters

`in`	*W,H*	new size

See Also

position(int,int), resize(int,int,int,int)

int Fl_Widget::take_focus ( )

Gives the widget the keyboard focus.

Tries to make this widget be the Fl::focus() widget, by first sending it an FL_FOCUS event, and if it returns non-zero, setting Fl::focus() to this widget. You should use this method to assign the focus to a widget.

Returns

true if the widget accepted the focus.

unsigned int Fl_Widget::takesevents ( ) const `[inline]`

Returns if the widget is able to take events.

This is the same as (active() && !output() && visible()) but is faster.

Return values

0	if the widget takes no events

int Fl_Widget::test_shortcut ( )

Returns true if the widget's label contains the entered '&x' shortcut.

This method must only be called in handle() methods or callbacks after a keypress event (usually FL_-KEYDOWN or FL_SHORTCUT). The widget's label is searched for a '&x' shortcut, and if found, this is compared with the entered key value.

Fl::event_text() is used to get the entered key value.

Returns

true, if the entered text matches the widget's'&x' shortcut, false (0) otherwise.

Note

Internal use only.

int Fl_Widget::test_shortcut (const char * *t*, const bool *require_alt* = `false`) `[static]`

Returns true if the given text `t` contains the entered '&x' shortcut.

This method must only be called in handle() methods or callbacks after a keypress event (usually FL_-KEYDOWN or FL_SHORTCUT). The given text `t` (usually a widget's label or menu text) is searched for a '&x' shortcut, and if found, this is compared with the entered key value.

Fl::event_text() is used to get the entered key value. Fl::event_state() is used to get the Alt modifier, if `require_alt` is true.

Parameters

t	text or label to search for '&x' shortcut.
require_alt	if true: match only if Alt key is pressed.

Returns

true, if the entered text matches the '&x' shortcut in `t` false (0) otherwise.

Note

Internal use only.

const char* Fl_Widget::tooltip ( ) const `[inline]`

Gets the current tooltip text.

Returns

a pointer to the tooltip text or NULL

See Also

tooltip(const char*), copy_tooltip(const char*)

void Fl_Widget::tooltip (const char * *text*)

Sets the current tooltip text.

Sets a string of text to display in a popup tooltip window when the user hovers the mouse over the widget. The string is *not* copied, so make sure any formatted string is stored in a static, global, or allocated buffer. If you want a copy made and managed for you, use the copy_tooltip() method, which will manage the tooltip string automatically.

If no tooltip is set, the tooltip of the parent is inherited. Setting a tooltip for a group and setting no tooltip for a child will show the group's tooltip instead. To avoid this behavior, you can set the child's tooltip to an empty string ("").

Parameters

in	*text*	New tooltip text (no copy is made)

See Also

copy_tooltip(const char∗), tooltip()

Fl_Window ∗ Fl_Widget::top_window ( ) const

Returns a pointer to the top-level window for the widget.
In other words, the 'window manager window' that contains this widget. This method differs from window() in that it won't return sub-windows (if there are any).

Returns

the top-level window, or NULL if no top-level window is associated with this widget.

See Also

window()

Fl_Window ∗ Fl_Widget::top_window_offset (int & *xoff,* int & *yoff*) const

Finds the x/y offset of the current widget relative to the top-level window.
Parameters

out	*xoff,yoff*	Returns the x/y offset

Returns

the top-level window (or NULL for a widget that's not in any window)

uchar Fl_Widget::type ( ) const [inline]

Gets the widget type.
Returns the widget type value, which is used for Forms compatibility and to simulate RTTI.

Todo Explain "simulate RTTI" (currently only used to decide if a widget is a window, i.e. type()>=FL_-WINDOW ?). Is type() really used in a way that ensures "Forms compatibility" ?

void Fl_Widget::type (uchar *t*) [inline]

Sets the widget type.
This is used for Forms compatibility.

void∗ Fl_Widget::user_data ( ) const [inline]

Gets the user data for this widget.
Gets the current user data (void ∗) argument that is passed to the callback function.

Returns

user data as a pointer

void Fl_Widget::user_data (void ∗ *v*) [inline]

Sets the user data for this widget.
Sets the new user data (void ∗) argument that is passed to the callback function.

Parameters

`in`	*v*	new user data

unsigned int Fl_Widget::visible ( ) const `[inline]`

Returns whether a widget is visible.
Return values

0	if the widget is not drawn and hence invisible.

See Also

show(), hide(), visible_r()

void Fl_Widget::visible_focus (int *v*) `[inline]`

Modifies keyboard focus navigation.
Parameters

`in`	*v*	set or clear visible focus

See Also

set_visible_focus(), clear_visible_focus(), visible_focus()

unsigned int Fl_Widget::visible_focus ( ) `[inline]`

Checks whether this widget has a visible focus.
Return values

0	if this widget has no visible focus.

See Also

visible_focus(int), set_visible_focus(), clear_visible_focus()

int Fl_Widget::visible_r ( ) const

Returns whether a widget and all its parents are visible.
Return values

0	if the widget or any of its parents are invisible.

See Also

show(), hide(), visible()

void Fl_Widget::w (int *v*) `[inline]`, `[protected]`

Internal use only.
Use position(int,int), size(int,int) or resize(int,int,int,int) instead.

int Fl_Widget::w ( ) const [inline]

Gets the widget width.

Returns

the width of the widget in pixels.

Fl_When Fl_Widget::when ( ) const [inline]

Returns the conditions under which the callback is called.
You can set the flags with when(uchar), the default value is FL_WHEN_RELEASE.

Returns

set of flags

See Also

when(uchar)

void Fl_Widget::when (uchar *i*) [inline]

Sets the flags used to decide when a callback is called.
This controls when callbacks are done. The following values are useful, the default value is FL_WHEN_RELEASE:

- 0: The callback is not done, but changed() is turned on.
- FL_WHEN_CHANGED: The callback is done each time the text is changed by the user.
- FL_WHEN_RELEASE: The callback will be done when this widget loses the focus, including when the window is unmapped. This is a useful value for text fields in a panel where doing the callback on every change is wasteful. However the callback will also happen if the mouse is moved out of the window, which means it should not do anything visible (like pop up an error message). You might do better setting this to zero, and scanning all the items for changed() when the OK button on a panel is pressed.
- FL_WHEN_ENTER_KEY: If the user types the Enter key, the entire text is selected, and the callback is done if the text has changed. Normally the Enter key will navigate to the next field (or insert a newline for a Fl_Multiline_Input) - this changes the behavior.
- FL_WHEN_ENTER_KEY|FL_WHEN_NOT_CHANGED: The Enter key will do the callback even if the text has not changed. Useful for command fields. Fl_Widget::when() is a set of bitflags used by subclasses of Fl_Widget to decide when to do the callback.

If the value is zero then the callback is never done. Other values are described in the individual widgets. This field is in the base class so that you can scan a panel and do_callback() on all the ones that don't do their own callbacks in response to an "OK" button.
Parameters

in	*i*	set of flags

Fl_Window ∗ Fl_Widget::window ( ) const

Returns a pointer to the nearest parent window up the widget hierarchy.
This will return sub-windows if there are any, or the parent window if there's no sub-windows. If this widget IS the top-level window, NULL is returned.

Return values

NULL	if no window is associated with this widget.

Note

for an Fl_Window widget, this returns its *parent* window (if any), not *this* window.

See Also

top_window()

void Fl_Widget::x (int *v*) `[inline]`, `[protected]`

Internal use only.
Use position(int,int), size(int,int) or resize(int,int,int,int) instead.

int Fl_Widget::x ( ) const `[inline]`

Gets the widget position in its window.

Returns

the x position relative to the window

void Fl_Widget::y (int *v*) `[inline]`, `[protected]`

Internal use only.
Use position(int,int), size(int,int) or resize(int,int,int,int) instead.

int Fl_Widget::y ( ) const `[inline]`

Gets the widget position in its window.

Returns

the y position relative to the window

The documentation for this class was generated from the following files:

- Fl_Widget.H
- Fl.cxx
- fl_boxtype.cxx
- fl_labeltype.cxx
- fl_shortcut.cxx
- Fl_Tooltip.cxx
- Fl_Widget.cxx
- Fl_Window.cxx

31.152 Fl_Widget_Tracker Class Reference

This class should be used to control safe widget deletion.
`#include <Fl.H>`

Public Member Functions

- int deleted ()

  Returns 1, if the watched widget has been deleted.
- int exists ()

  Returns 1, if the watched widget exists (has not been deleted).
- Fl_Widget_Tracker (Fl_Widget ∗wi)

  The constructor adds a widget to the watch list.
- Fl_Widget ∗ widget ()

  Returns a pointer to the watched widget.
- ∼Fl_Widget_Tracker ()

  The destructor removes a widget from the watch list.

31.152.1 Detailed Description

This class should be used to control safe widget deletion.

You can use an Fl_Widget_Tracker object to watch another widget, if you need to know, if this widget has been deleted during a callback.

This simplifies the use of the "safe widget deletion" methods Fl::watch_widget_pointer() and Fl::release-_widget_pointer() and makes their use more reliable, because the destructor autmatically releases the widget pointer from the widget watch list.

It is intended to be used as an automatic (local/stack) variable, such that the automatic destructor is called when the object's scope is left. This ensures that no stale widget pointers are left in the widget watch list (see example below).

You can also create Fl_Widget_Tracker objects with `new`, but then it is your responsibility to delete the object (and thus remove the widget pointer from the watch list) when it is not needed any more.

Example:

```
int MyClass::handle (int event) {

  if (...) {
    Fl_Widget_Tracker wp(this);       // watch myself
    do_callback();                     // call the callback

    if (wp.deleted()) return 1;        // exit, if deleted

    // Now we are sure that the widget has not been deleted.
    // It is safe to access the widget

    clear_changed();          // access the widget
  }
}
```

31.152.2 Member Function Documentation

int Fl_Widget_Tracker::deleted ( ) [inline]

Returns 1, if the watched widget has been deleted.

This is a convenience method. You can also use something like

`if (wp.widget() == 0) // ...`

where `wp` is an Fl_Widget_Tracker object.

int Fl_Widget_Tracker::exists ( ) [inline]

Returns 1, if the watched widget exists (has not been deleted).

This is a convenience method. You can also use something like

`if (wp.widget() != 0) // ...`

where `wp` is an Fl_Widget_Tracker object.

Fl_Widget∗ Fl_Widget_Tracker::widget ( ) `[inline]`

Returns a pointer to the watched widget.

This pointer is `NULL`, if the widget has been deleted.

The documentation for this class was generated from the following files:

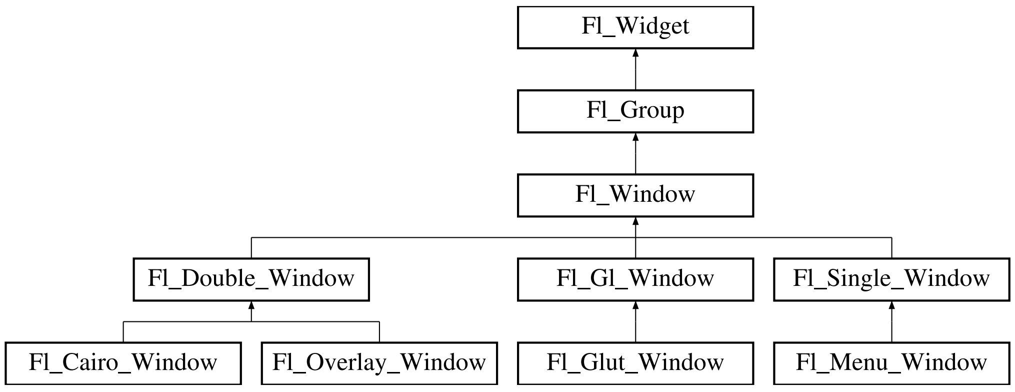
- Fl.H
- Fl.cxx

31.153 Fl_Window Class Reference

This widget produces an actual window.

`#include <Fl_Window.H>`

Inheritance diagram for Fl_Window:



Classes

- struct shape_data_type

  Data supporting a non-rectangular window shape.

Public Member Functions

- virtual Fl_Window ∗ as_window ()

  Returns an Fl_Window pointer if this widget is an Fl_Window.
- void border (int b)

  Sets whether or not the window manager border is around the window.
- unsigned int border () const

  See void Fl_Window::border(int)
- void clear_border ()

  Fast inline function to turn the window manager border off.
- void clear_modal_states ()

  Clears the "modal" flags and converts a "modal" or "non-modal" window back into a "normal" window.
- void copy_label (const char ∗a)

  Sets the window titlebar label to a copy of a character string.
- void cursor (Fl_Cursor)

  Changes the cursor for this window.
- void cursor (const Fl_RGB_Image ∗, int, int)

Changes the cursor for this window.
- void cursor (Fl_Cursor c, Fl_Color, Fl_Color=FL_WHITE)

  For back compatibility only.
- int decorated_h ()

  Returns the window height including any window title bar and any frame added by the window manager.
- int decorated_w ()

  Returns the window width including any frame added by the window manager.
- void default_cursor (Fl_Cursor)

  Sets the default window cursor.
- void default_cursor (Fl_Cursor c, Fl_Color, Fl_Color=FL_WHITE)

  For back compatibility only.
- Fl_Window (int w, int h, const char ∗title=0)

  Creates a window from the given size and title.
- Fl_Window (int x, int y, int w, int h, const char ∗title=0)

  Creates a window from the given position, size and title.
- void free_position ()

  Undoes the effect of a previous resize() or show() so that the next time show() is called the window manager is free to position the window.
- void fullscreen ()

  Makes the window completely fill one or more screens, without any window manager border visible.
- unsigned int fullscreen_active () const

  Returns non zero if FULLSCREEN flag is set, 0 otherwise.
- void fullscreen_off ()

  Turns off any side effects of fullscreen()
- void fullscreen_off (int X, int Y, int W, int H)

  Turns off any side effects of fullscreen() and does resize(x,y,w,h).
- void fullscreen_screens (int top, int bottom, int left, int right)

  Sets which screens should be used when this window is in fullscreen mode.
- virtual int handle (int)

  Handles the specified event.
- virtual void hide ()

  Removes the window from the screen.
- void hotspot (int x, int y, int offscreen=0)

  Positions the window so that the mouse is pointing at the given position, or at the center of the given widget, which may be the window itself.
- void hotspot (const Fl_Widget ∗, int offscreen=0)

  See void Fl_Window::hotspot(int x, int y, int offscreen = 0)
- void hotspot (const Fl_Widget &p, int offscreen=0)

  See void Fl_Window::hotspot(int x, int y, int offscreen = 0)
- void icon (const Fl_RGB_Image ∗)

  Sets or resets a single window icon.
- const void ∗ icon () const

  Gets the current icon window target dependent data.
- void icon (const void ∗ic)

  Sets the current icon window target dependent data.
- void iconize ()

  Iconifies the window.

- const char ∗ iconlabel () const

  See void Fl_Window::iconlabel(const char∗)
- void iconlabel (const char ∗)

  Sets the icon label.
- void icons (const Fl_RGB_Image ∗[ ], int)

  Sets the window icons.
- const char ∗ label () const

  See void Fl_Window::label(const char∗)
- void label (const char ∗)

  Sets the window title bar label.
- void label (const char ∗label, const char ∗iconlabel)

  Sets the icon label.
- void make_current ()

  Sets things up so that the drawing functions in <FL/fl_draw.H> will go into this window.
- unsigned int menu_window () const

  Returns true if this window is a menu window.
- unsigned int modal () const

  Returns true if this window is modal.
- unsigned int non_modal () const

  Returns true if this window is modal or non-modal.
- unsigned int override () const

  Returns non zero if FL_OVERRIDE flag is set, 0 otherwise.
- virtual void resize (int X, int Y, int W, int H)

  Changes the size and position of the window.
- void set_menu_window ()

  Marks the window as a menu window.
- void set_modal ()

  A "modal" window, when shown(), will prevent any events from being delivered to other windows in the same program, and will also remain on top of the other windows (if the X window manager supports the "transient for" property).
- void set_non_modal ()

  A "non-modal" window (terminology borrowed from Microsoft Windows) acts like a modal() one in that it remains on top, but it has no effect on event delivery.
- void set_override ()

  Activates the flags NOBORDER|FL_OVERRIDE.
- void set_tooltip_window ()

  Marks the window as a tooltip window.
- void shape (const Fl_Image ∗img)

  Assigns a non-rectangular shape to the window.
- void shape (const Fl_Image &b)

  Set the window's shape with an Fl_Image.
- virtual void show ()

  Puts the window on the screen.
- void show (int argc, char ∗∗argv)

  Puts the window on the screen and parses command-line arguments.
- int shown ()

  Returns non-zero if show() has been called (but not hide()).

- void size_range (int minw, int minh, int maxw=0, int maxh=0, int dw=0, int dh=0, int aspect=0)

  Sets the allowable range the user can resize this window to.
- unsigned int tooltip_window () const

  Returns true if this window is a tooltip window.
- void wait_for_expose ()

  Waits for the window to be displayed after calling show().
- int x_root () const

  Gets the x position of the window on the screen.
- const char ∗ xclass () const

  Returns the xclass for this window, or a default.
- void xclass (const char ∗c)

  Sets the xclass for this window.
- int y_root () const

  Gets the y position of the window on the screen.
- virtual ∼Fl_Window ()

  The destructor also deletes all the children.

Static Public Member Functions

- static Fl_Window ∗ current ()

  Returns the last window that was made current.
- static void default_callback (Fl_Window ∗, void ∗v)

  Back compatibility: Sets the default callback v for win to call on close event.
- static void default_icon (const Fl_RGB_Image ∗)

  Sets a single default window icon.
- static void default_icons (const Fl_RGB_Image ∗[ ], int)

  Sets the default window icons.
- static void default_xclass (const char ∗)

  Sets the default window xclass.
- static const char ∗ default_xclass ()

  Returns the default xclass.

Protected Member Functions

- virtual void draw ()

  Draws the widget.
- virtual void flush ()

  Forces the window to be drawn, this window is also made current and calls draw().
- void force_position (int force)

  Sets an internal flag that tells FLTK and the window manager to honor position requests.
- int force_position () const

  Returns the internal state of the window's FORCE_POSITION flag.
- void free_icons ()

  Deletes all icons previously attached to the window.

Protected Attributes

- shape_data_type ∗ shape_data_

  non-null means the window has a non-rectangular shape

Static Protected Attributes

- static Fl_Window ∗ current_

  Stores the last window that was made current.

Friends

- class **Fl_X**

Additional Inherited Members

31.153.1 Detailed Description

This widget produces an actual window.

This can either be a main window, with a border and title and all the window management controls, or a "subwindow" inside a window. This is controlled by whether or not the window has a parent().

Once you create a window, you usually add children Fl_Widget 's to it by using window->add(child) for each new widget. See Fl_Group for more information on how to add and remove children.

There are several subclasses of Fl_Window that provide double-buffering, overlay, menu, and OpenGL support.

The window's callback is done if the user tries to close a window using the window manager and Fl::modal() is zero or equal to the window. Fl_Window has a default callback that calls Fl_Window::hide().

31.153.2 Constructor & Destructor Documentation

Fl_Window::Fl_Window (int *w*, int *h*, const char ∗ *title* = 0)

Creates a window from the given size and title.

If Fl_Group::current() is not NULL, the window is created as a subwindow of the parent window.

The (w,h) form of the constructor creates a top-level window and asks the window manager to position the window. The (x,y,w,h) form of the constructor either creates a subwindow or a top-level window at the specified location (x,y) , subject to window manager configuration. If you do not specify the position of the window, the window manager will pick a place to show the window or allow the user to pick a location. Use position(x,y) or hotspot() before calling show() to request a position on the screen. See Fl_Window::resize() for some more details on positioning windows.

Top-level windows initially have visible() set to 0 and parent() set to NULL. Subwindows initially have visible() set to 1 and parent() set to the parent window pointer.

Fl_Widget::box() defaults to FL_FLAT_BOX. If you plan to completely fill the window with children widgets you should change this to FL_NO_BOX. If you turn the window border off you may want to change this to FL_UP_BOX.

See Also

Fl_Window(int x, int y, int w, int h, const char∗ title)

Fl_Window::Fl_Window (int *x*, int *y*, int *w*, int *h*, const char ∗ *title* = 0)

Creates a window from the given position, size and title.

See Also

Fl_Window(int w, int h, const char ∗title)

Fl_Window::∼Fl_Window ( ) [virtual]

The destructor *also deletes all the children*.

This allows a whole tree to be deleted at once, without having to keep a pointer to all the children in the user code. A kludge has been done so the Fl_Window and all of its children can be automatic (local) variables, but you must declare the Fl_Window *first* so that it is destroyed last.

31.153.3 Member Function Documentation

virtual Fl_Window∗ Fl_Window::as_window ( ) [inline], [virtual]

Returns an Fl_Window pointer if this widget is an Fl_Window.

Use this method if you have a widget (pointer) and need to know whether this widget is derived from Fl_Window. If it returns non-NULL, then the widget in question is derived from Fl_Window, and you can use the returned pointer to access its children or other Fl_Window-specific methods.

Return values

NULL	if this widget is not derived from Fl_Window.

Note

This method is provided to avoid dynamic_cast.

See Also

Fl_Widget::as_group(), Fl_Widget::as_gl_window()

Reimplemented from Fl_Widget.

void Fl_Window::border (int *b*)

Sets whether or not the window manager border is around the window.

The default value is true. void border(int) can be used to turn the border on and off. *Under most X window managers this does not work after show() has been called, although SGI's 4DWM does work.*

void Fl_Window::clear_border ( ) [inline]

Fast inline function to turn the window manager border off.

It only works before show() is called.

void Fl_Window::clear_modal_states ( ) [inline]

Clears the "modal" flags and converts a "modal" or "non-modal" window back into a "normal" window.

Note that there are *three* states for a window: modal, non-modal, and normal.

You can not change the "modality" of a window whilst it is shown, so it is necessary to first hide() the window, change its "modality" as required, then re-show the window for the new state to take effect.

This method can also be used to change a "modal" window into a "non-modal" one. On several supported platforms, the "modal" state over-rides the "non-modal" state, so the "modal" state must be cleared before the window can be set into the "non-modal" state. In general, the following sequence should work:

```
win->hide();
win->clear_modal_states();
//Set win to new state as desired, or leave "normal", e.g...
win->set_non_modal();
win->show();
```

Note

> Under some window managers, the sequence of hiding the window and changing its modality will often cause it to be re-displayed at a different position when it is subsequently shown. This is an irritating feature but appears to be unavoidable at present. As a result we would advise to use this method only when absolutely necessary.

See Also

> void set_modal(), void set_non_modal()

Fl_Window ∗ Fl_Window::current ( ) `[static]`

Returns the last window that was made current.

See Also

> Fl_Window::make_current()

void Fl_Window::cursor (Fl_Cursor *c*)

Changes the cursor for this window.

This always calls the system, if you are changing the cursor a lot you may want to keep track of how you set it in a static variable and call this only if the new cursor is different.

The type Fl_Cursor is an enumeration defined in <FL/Enumerations.H>.

See Also

> cursor(const Fl_RGB_Image∗, int, int), default_cursor()

void Fl_Window::cursor (const Fl_RGB_Image ∗ *image,* int *hotx,* int *hoty*)

Changes the cursor for this window.

This always calls the system, if you are changing the cursor a lot you may want to keep track of how you set it in a static variable and call this only if the new cursor is different.

The default cursor will be used if the provided image cannot be used as a cursor.

See Also

> cursor(Fl_Cursor), default_cursor()

void Fl_Window::cursor (Fl_Cursor *c,* Fl_Color *,* Fl_Color = *`FL_WHITE`*)

For back compatibility only.

Same as Fl_Window::cursor(Fl_Cursor)

int Fl_Window::decorated_h ( )

Returns the window height including any window title bar and any frame added by the window manager.

Same as h() if applied to a subwindow.

int Fl_Window::decorated_w ( )

Returns the window width including any frame added by the window manager.

Same as w() if applied to a subwindow.

void Fl_Window::default_cursor (Fl_Cursor *c*)

Sets the default window cursor.

This is the cursor that will be used after the mouse pointer leaves a widget with a custom cursor set.

See Also

cursor(const Fl_RGB_Image∗, int, int), default_cursor()

void Fl_Window::default_cursor (Fl_Cursor *c*, Fl_Color , Fl_Color = *FL_WHITE*)

For back compatibility only.

same as Fl_Window::default_cursor(Fl_Cursor)

void Fl_Window::default_icon (const Fl_RGB_Image ∗ *icon*) `[static]`

Sets a single default window icon.

If `icon` is NULL the current default icons are removed.

Parameters

in	*icon*	default icon for all windows subsequently created or NULL

See Also

Fl_Window::default_icons(const Fl_RGB_Image ∗[ ], int)
Fl_Window::icon(const Fl_RGB_Image ∗)
Fl_Window::icons(const Fl_RGB_Image ∗[ ], int)

void Fl_Window::default_icons (const Fl_RGB_Image ∗ *icons[ ]*, int *count*) `[static]`

Sets the default window icons.

The default icons are used for all windows that don't have their own icons set before show() is called. You can change the default icons whenever you want, but this only affects windows that are created (and shown) after this call.

The given images in `icons` are copied. You can use a local variable or free the images immediately after this call.

Parameters

in	*icons*	default icons for all windows subsequently created
in	*count*	number of images in `icons`. Set to 0 to remove the current default icons

See Also

Fl_Window::default_icon(const Fl_RGB_Image ∗)
Fl_Window::icon(const Fl_RGB_Image ∗)
Fl_Window::icons(const Fl_RGB_Image ∗[ ], int)

void Fl_Window::default_xclass (const char ∗ *xc*) `[static]`

Sets the default window xclass.

The default xclass is used for all windows that don't have their own xclass set before show() is called. You can change the default xclass whenever you want, but this only affects windows that are created (and shown) after this call.

The given string `xc` is copied. You can use a local variable or free the string immediately after this call.

If you don't call this, the default xclass for all windows will be "FLTK". You can reset the default xclass by specifying NULL for `xc`.

If you call Fl_Window::xclass(const char ∗) for any window, then this also sets the default xclass, unless it has been set before.

Parameters

in	*xc*	default xclass for all windows subsequently created

See Also

Fl_Window::xclass(const char ∗)

const char ∗ Fl_Window::default_xclass ( ) `[static]`

Returns the default xclass.

See Also

Fl_Window::default_xclass(const char ∗)

void Fl_Window::draw ( ) `[protected]`, `[virtual]`

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call redraw() instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw() method*, e.g. for an embedded scrollbar, you can do it (because draw() is virtual) like this:

```
Fl_Widget *s = &scroll;              // scroll is an embedded Fl_Scrollbar
s->draw();                 // calls Fl_Scrollbar::draw()
```

Reimplemented from Fl_Group.

Reimplemented in Fl_Gl_Window, Fl_Cairo_Window, and Fl_Glut_Window.

void Fl_Window::flush ( ) `[protected]`, `[virtual]`

Forces the window to be drawn, this window is also made current and calls draw().

Reimplemented in Fl_Gl_Window, Fl_Overlay_Window, Fl_Double_Window, Fl_Single_Window, and Fl_Menu_Window.

void Fl_Window::force_position (int *force*) `[inline]`, `[protected]`

Sets an internal flag that tells FLTK and the window manager to honor position requests.

This is used internally and should not be needed by user code.

Parameters

in	*force*	1 to set the FORCE_POSITION flag, 0 to clear it

int Fl_Window::force_position ( ) const `[inline]`, `[protected]`

Returns the internal state of the window's FORCE_POSITION flag.

Return values

1	if flag is set

0	otherwise

See Also

force_position(int)

void Fl_Window::free_icons ( ) `[protected]`

Deletes all icons previously attached to the window.

See Also

Fl_Window::icons(const Fl_RGB_Image ∗icons[ ], int count)

void Fl_Window::free_position ( ) `[inline]`

Undoes the effect of a previous resize() or show() so that the next time show() is called the window manager is free to position the window.

This is for Forms compatibility only.

Deprecated please use force_position(0) instead

void Fl_Window::fullscreen ( )

Makes the window completely fill one or more screens, without any window manager border visible.

You must use fullscreen_off() to undo this.

Note

On some platforms, this can result in the keyboard being grabbed. The window may also be recreated, meaning hide() and show() will be called.

See Also

void Fl_Window::fullscreen_screens()

void Fl_Window::fullscreen_screens (int *top*, int *bottom*, int *left*, int *right*)

Sets which screens should be used when this window is in fullscreen mode.

The window will be resized to the top of the screen with index `top`, the bottom of the screen with index `bottom`, etc.

If this method is never called, or if any argument is < 0, then the window will be resized to fill the screen it is currently on.

See Also

void Fl_Window::fullscreen()

int Fl_Window::handle (int *event*) `[virtual]`

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited handle() method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters

in	*event*	the kind of event received

Return values

0	if the event was not used or understood
1	if the event was used and can be deleted

See Also

Fl_Event

Reimplemented from Fl_Group.
Reimplemented in Fl_Gl_Window, and Fl_Glut_Window.

void Fl_Window::hide ( ) `[virtual]`

Removes the window from the screen.
If the window is already hidden or has not been shown then this does nothing and is harmless.
Reimplemented from Fl_Widget.
Reimplemented in Fl_Gl_Window, Fl_Overlay_Window, Fl_Double_Window, and Fl_Menu_Window.

void Fl_Window::hotspot (int *x*, int *y*, int *offscreen* = `0`)

Positions the window so that the mouse is pointing at the given position, or at the center of the given widget, which may be the window itself.
If the optional offscreen parameter is non-zero, then the window is allowed to extend off the screen (this does not work with some X window managers).

See Also

position()

void Fl_Window::icon (const Fl_RGB_Image ∗ *icon*)

Sets or resets a single window icon.
A window icon *can* be changed while the window is shown, but this *may* be platform and/or window manager dependent. To be sure that the window displays the correct window icon you should always set the icon before the window is shown.
If a window icon has not been set for a particular window, then the default window icon (see links below) or the system default icon will be used.
Parameters

in	*icon*	icon for this window, NULL to reset window icon.

See Also

Fl_Window::default_icon(const Fl_RGB_Image ∗)
Fl_Window::default_icons(const Fl_RGB_Image ∗[ ], int)
Fl_Window::icons(const Fl_RGB_Image ∗[ ], int)

const void ∗ Fl_Window::icon ( ) const

Gets the current icon window target dependent data.

Deprecated in 1.3.3

void Fl_Window::icon (const void ∗ *ic*)

Sets the current icon window target dependent data.

Deprecated in 1.3.3

void Fl_Window::iconize ( )

Iconifies the window.

If you call this when shown() is false it will show() it as an icon. If the window is already iconified this does nothing.

Call show() to restore the window.

When a window is iconified/restored (either by these calls or by the user) the handle() method is called with FL_HIDE and FL_SHOW events and visible() is turned on and off.

There is no way to control what is drawn in the icon except with the string passed to Fl_Window::xclass(). You should not rely on window managers displaying the icons.

void Fl_Window::iconlabel (const char ∗ *iname*)

Sets the icon label.

void Fl_Window::icons (const Fl_RGB_Image ∗ *icons[ ],* int *count*)

Sets the window icons.

You may set multiple window icons with different sizes. Dependent on the platform and system settings the best (or the first) icon will be chosen.

The given images in `icons` are copied. You can use a local variable or free the images immediately after this call.

If `count` is zero, current icons are removed. If `count` is greater than zero (must not be negative), then `icons`[] must contain at least `count` valid image pointers (not NULL). Otherwise the behavior is undefined.

Parameters

in	*icons*	icons for this window
in	*count*	number of images in `icons`. Set to 0 to remove the current icons

See Also

Fl_Window::default_icon(const Fl_RGB_Image ∗)
Fl_Window::default_icons(const Fl_RGB_Image ∗[ ], int)
Fl_Window::icon(const Fl_RGB_Image ∗)

void Fl_Window::label (const char ∗ *name*)

Sets the window title bar label.

void Fl_Window::label (const char ∗ *label,* const char ∗ *iconlabel*)

Sets the icon label.

void Fl_Window::make_current ( )

Sets things up so that the drawing functions in <FL/fl_draw.H> will go into this window.

This is useful for incremental update of windows, such as in an idle callback, which will make your program behave much better if it draws a slow graphic. **Danger: incremental update is very hard to debug and maintain!**

This method only works for the Fl_Window and Fl_Gl_Window derived classes.

unsigned int Fl_Window::menu_window ( ) const [inline]

Returns true if this window is a menu window.

unsigned int Fl_Window::modal ( ) const [inline]

Returns true if this window is modal.

unsigned int Fl_Window::non_modal ( ) const [inline]

Returns true if this window is modal or non-modal.

unsigned int Fl_Window::override ( ) const [inline]

Returns non zero if FL_OVERRIDE flag is set, 0 otherwise.

virtual void Fl_Window::resize (int *X*, int *Y*, int *W*, int *H*) [virtual]

Changes the size and position of the window.

If shown() is true, these changes are communicated to the window server (which may refuse that size and cause a further resize). If shown() is false, the size and position are used when show() is called. See Fl_Group for the effect of resizing on the child widgets.

You can also call the Fl_Widget methods size(x,y) and position(w,h), which are inline wrappers for this virtual function.

A top-level window can not force, but merely suggest a position and size to the operating system. The window manager may not be willing or able to display a window at the desired position or with the given dimensions. It is up to the application developer to verify window parameters after the resize request.

Reimplemented from Fl_Group.

Reimplemented in Fl_Gl_Window, Fl_Overlay_Window, and Fl_Double_Window.

void Fl_Window::set_menu_window ( ) [inline]

Marks the window as a menu window.

This is intended for internal use, but it can also be used if you write your own menu handling. However, this is not recommended.

This flag is used for correct ”parenting” of windows in communication with the windowing system. Modern X window managers can use different flags to distinguish menu and tooltip windows from normal windows.

This must be called before the window is shown and cannot be changed later.

void Fl_Window::set_modal ( ) [inline]

A ”modal” window, when shown(), will prevent any events from being delivered to other windows in the same program, and will also remain on top of the other windows (if the X window manager supports the ”transient for” property).

Several modal windows may be shown at once, in which case only the last one shown gets events. You can see which window (if any) is modal by calling Fl::modal().

void Fl_Window::set_non_modal ( ) [inline]

A ”non-modal” window (terminology borrowed from Microsoft Windows) acts like a modal() one in that it remains on top, but it has no effect on event delivery.

There are *three* states for a window: modal, non-modal, and normal.

void Fl_Window::set_tooltip_window ( ) `[inline]`

Marks the window as a tooltip window.

This is intended for internal use, but it can also be used if you write your own tooltip handling. However, this is not recommended.

This flag is used for correct "parenting" of windows in communication with the windowing system. Modern X window managers can use different flags to distinguish menu and tooltip windows from normal windows.

This must be called before the window is shown and cannot be changed later.

Note

> Since Fl_Tooltip_Window is derived from Fl_Menu_Window, this also **clears** the menu_window() state.

void Fl_Window::shape (const Fl_Image * *img*)

Assigns a non-rectangular shape to the window.

This function gives an arbitrary shape (not just a rectangular region) to an Fl_Window. An Fl_Image of any dimension can be used as mask; it is rescaled to the window's dimension as needed.

The layout and widgets inside are unaware of the mask shape, and most will act as though the window's rectangular bounding box is available to them. It is up to you to make sure they adhere to the bounds of their masking shape.

The `img` argument can be an Fl_Bitmap, Fl_Pixmap, Fl_RGB_Image or Fl_Shared_Image:

- With Fl_Bitmap or Fl_Pixmap, the shaped window covers the image part where bitmap bits equal one, or where the pixmap is not fully transparent.
- With an Fl_RGB_Image with an alpha channel (depths 2 or 4), the shaped window covers the image part that is not fully transparent.
- With an Fl_RGB_Image of depth 1 (gray-scale) or 3 (RGB), the shaped window covers the non-black image part.
- With an Fl_Shared_Image, the shape is determined by rules above applied to the underlying image. The shared image should not have been scaled through Fl_Shared_Image::scale().

Platform details:

- On the unix/linux platform, the SHAPE extension of the X server is required. This function does control the shape of Fl_Gl_Window instances.
- On the MSWindows platform, this function does nothing with class Fl_Gl_Window.
- On the Mac platform, OS version 10.4 or above is required. An 8-bit shape-mask is used when `img` is an Fl_RGB_Image: with depths 2 or 4, the image alpha channel becomes the shape mask such that areas with alpha = 0 are out of the shaped window; with depths 1 or 3, white and black are in and out of the shaped window, respectively, and other colors give intermediate masking scores. This function does nothing with class Fl_Gl_Window.

The window borders and caption created by the window system are turned off by default. They can be re-enabled by calling Fl_Window::border(1).

A usage example is found at example/shapedwindow.cxx.

Version

> 1.3.3 (and requires compilation with FLTK_ABI_VERSION >= 10303)

void Fl_Window::shape (const Fl_Image & *b*) `[inline]`

Set the window's shape with an Fl_Image.

See Also

void shape(const Fl_Image∗ img)

virtual void Fl_Window::show ( ) `[virtual]`

Puts the window on the screen.

Usually (on X) this has the side effect of opening the display.

If the window is already shown then it is restored and raised to the top. This is really convenient because your program can call show() at any time, even if the window is already up. It also means that show() serves the purpose of raise() in other toolkits.

Fl_Window::show(int argc, char ∗∗argv) is used for top-level windows and allows standard arguments to be parsed from the command-line.

Note

For some obscure reasons Fl_Window::show() resets the current group by calling Fl_Group::current(0). The comments in the code say "get rid of very common user bug: forgot end()". Although this is true it may have unwanted side effects if you show() an unrelated window (maybe for an error message or warning) while building a window or any other group widget.

Todo Check if we can remove resetting the current group in a later FLTK version (after 1.3.x). This may break "already broken" programs though if they rely on this "feature".

See Also

Fl_Window::show(int argc, char ∗∗argv)

Reimplemented from Fl_Widget.

Reimplemented in Fl_Gl_Window, Fl_Overlay_Window, Fl_Double_Window, Fl_Single_Window, and Fl_Menu_Window.

void Fl_Window::show (int *argc*, char ∗∗ *argv*)

Puts the window on the screen and parses command-line arguments.

Usually (on X) this has the side effect of opening the display.

This form should be used for top-level windows, at least for the first (main) window. It allows standard arguments to be parsed from the command-line. You can use `argc` and `argv` from main(int argc, char ∗∗argv) for this call.

The first call also sets up some system-specific internal variables like the system colors.

Todo explain which system parameters are set up.

Parameters

argc	command-line argument count, usually from main()
argv	command-line argument vector, usually from main()

See Also

virtual void Fl_Window::show()

int Fl_Window::shown ( ) `[inline]`

Returns non-zero if show() has been called (but not hide()).

You can tell if a window is iconified with (w->shown() && !w->visible()).

void Fl_Window::size_range (int *minw*, int *minh*, int *maxw* = *0*, int *maxh* = *0*, int *dw* = *0*, int *dh* = *0*, int *aspect* = *0*) `[inline]`

Sets the allowable range the user can resize this window to.

This only works for top-level windows.

- `minw` and `minh` are the smallest the window can be. Either value must be greater than 0.
- `maxw` and `maxh` are the largest the window can be. If either is *equal* to the minimum then you cannot resize in that direction. If either is zero then FLTK picks a maximum size in that direction such that the window will fill the screen.
- `dw` and `dh` are size increments. The window will be constrained to widths of minw + N $*$ dw, where N is any non-negative integer. If these are less or equal to 1 they are ignored (this is ignored on WIN32).
- `aspect` is a flag that indicates that the window should preserve its aspect ratio. This only works if both the maximum and minimum have the same aspect ratio (ignored on WIN32 and by many X window managers).

If this function is not called, FLTK tries to figure out the range from the setting of resizable():

- If resizable() is NULL (this is the default) then the window cannot be resized and the resize border and max-size control will not be displayed for the window.
- If either dimension of resizable() is less than 100, then that is considered the minimum size. Otherwise the resizable() has a minimum size of 100.
- If either dimension of resizable() is zero, then that is also the maximum size (so the window cannot resize in that direction).

It is undefined what happens if the current size does not fit in the constraints passed to size_range().

unsigned int Fl_Window::tooltip_window ( ) const `[inline]`

Returns true if this window is a tooltip window.

void Fl_Window::wait_for_expose ( )

Waits for the window to be displayed after calling show().

Fl_Window::show() is not guaranteed to show and draw the window on all platforms immediately. Instead this is done in the background; particularly on X11 it will take a few messages (client server roundtrips) to display the window. Usually this small delay doesn't matter, but in some cases you may want to have the window instantiated and displayed synchronously.

Currently (as of FLTK 1.3.4) this method has an effect on X11 and Mac OS. On Windows, show() is always synchronous. The effect of show() varies with versions of Mac OS X: early versions have the window appear on the screen when show() returns, later versions don't. If you want to write portable code and need this synchronous show() feature, add win->wait_for_expose() on all platforms, and FLTK will just do the right thing.

This method can be used for displaying splash screens before calling Fl::run() or for having exact control over which window has the focus after calling show().

If the window is not shown(), this method does nothing.

Note

> Depending on the platform and window manager wait_for_expose() may not guarantee that the window is fully drawn when it is called. Under X11 it may only make sure that the window is **mapped**, i.e. the internal (OS dependent) window object was created (and maybe shown on the desktop as an empty frame or something like that). You may need to call Fl::flush() after wait_for_expose() to make sure the window and all its widgets are drawn and thus visible.
> FLTK does the best it can do to make sure that all widgets get drawn if you call wait_for_expose() and Fl::flush(). However, dependent on the window manager it can not be guaranteed that this does always happen synchronously. The only guaranteed behavior that all widgets are eventually drawn is if the FLTK event loop is run continuously, for instance with Fl::run().

See Also

> virtual void Fl_Window::show()

Example code for displaying a window before calling Fl::run()

```
Fl_Double_Window win = new Fl_Double_Window(...);

// do more window initialization here ...

win->show();                // show window
win->wait_for_expose();    // wait, until displayed
Fl::flush();                // make sure everything gets drawn

// do more initialization work that needs some time here ...

Fl::run();                  // start FLTK event loop
```

Note that the window will not be responsive until the event loop is started with Fl::run().

const char ∗ Fl_Window::xclass ( ) const

Returns the xclass for this window, or a default.

See Also

> Fl_Window::default_xclass(const char ∗)
> Fl_Window::xclass(const char ∗)

void Fl_Window::xclass (const char ∗ *xc*)

Sets the xclass for this window.

A string used to tell the system what type of window this is. Mostly this identifies the picture to draw in the icon. This only works if called *before* calling show().

Under X, this is turned into a XA_WM_CLASS pair by truncating at the first non-alphanumeric character and capitalizing the first character, and the second one if the first is 'x'. Thus "foo" turns into "foo, Foo", and "xprog.1" turns into "xprog, XProg".

Under Microsoft Windows, this string is used as the name of the WNDCLASS structure, though it is not clear if this can have any visible effect.

Since

> FLTK 1.3 the passed string is copied. You can use a local variable or free the string immediately after this call. Note that FLTK 1.1 stores the *pointer* without copying the string.

If the default xclass has not yet been set, this also sets the default xclass for all windows created subsequently.

See Also

> Fl_Window::default_xclass(const char ∗)

31.153.4 Member Data Documentation

Fl_Window∗ Fl_Window::current_ `[static], [protected]`

Stores the last window that was made current.

See current() const

The documentation for this class was generated from the following files:

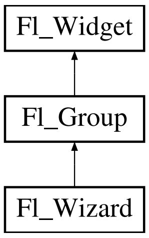
- Fl_Window.H
- Fl.cxx
- Fl_arg.cxx
- fl_cursor.cxx
- Fl_Window.cxx
- Fl_Window_fullscreen.cxx
- Fl_Window_hotspot.cxx
- Fl_Window_iconize.cxx
- Fl_Window_shape.cxx

31.154 Fl_Wizard Class Reference

This widget is based off the Fl_Tabs widget, but instead of displaying tabs it only changes "tabs" under program control.

`#include <Fl_Wizard.H>`

Inheritance diagram for Fl_Wizard:



Public Member Functions

- Fl_Wizard (int, int, int, int, const char ∗=0)

  The constructor creates the Fl_Wizard widget at the specified position and size.
- void next ()

  This method shows the next child of the wizard.
- void prev ()

  Shows the previous child.
- Fl_Widget ∗ value ()

  Gets the current visible child widget.
- void value (Fl_Widget ∗)

  Sets the child widget that is visible.

Additional Inherited Members

31.154.1 Detailed Description

This widget is based off the Fl_Tabs widget, but instead of displaying tabs it only changes "tabs" under program control.

Its primary purpose is to support "wizards" that step a user through configuration or troubleshooting tasks.

As with Fl_Tabs, wizard panes are composed of child (usually Fl_Group) widgets. Navigation buttons must be added separately.

31.154.2 Constructor & Destructor Documentation

Fl_Wizard::Fl_Wizard (int ***xx,*** **int** ***yy,*** **int** ***ww,*** **int** ***hh,*** **const char** ∗ ***l*** = `0` **)**

The constructor creates the Fl_Wizard widget at the specified position and size.

The inherited destructor destroys the widget and its children.

31.154.3 Member Function Documentation

void Fl_Wizard::next ( )

This method shows the next child of the wizard.

If the last child is already visible, this function does nothing.

void Fl_Wizard::prev ( )

Shows the previous child.

Fl_Widget ∗ **Fl_Wizard::value ( )**

Gets the current visible child widget.

void Fl_Wizard::value (Fl_Widget ∗ ***kid*** **)**

Sets the child widget that is visible.

The documentation for this class was generated from the following files:

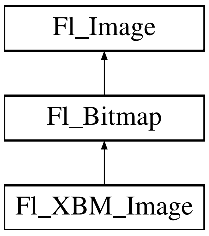
- Fl_Wizard.H
- Fl_Wizard.cxx

31.155 Fl_XBM_Image Class Reference

The Fl_XBM_Image class supports loading, caching, and drawing of X Bitmap (XBM) bitmap files.

`#include <Fl_XBM_Image.H>`

Inheritance diagram for Fl_XBM_Image:



Public Member Functions

- Fl_XBM_Image (const char ∗filename)

    The constructor loads the named XBM file from the given name filename.

Additional Inherited Members

31.155.1 Detailed Description

The Fl_XBM_Image class supports loading, caching, and drawing of X Bitmap (XBM) bitmap files.

31.155.2 Constructor & Destructor Documentation

Fl_XBM_Image::Fl_XBM_Image (const char ∗ *name*)

The constructor loads the named XBM file from the given name filename.

The destructor frees all memory and server resources that are used by the image.

The documentation for this class was generated from the following files:

- Fl_XBM_Image.H
- Fl_XBM_Image.cxx

31.156 Fl_XColor Struct Reference

Public Attributes

- unsigned char **b**
- unsigned char **g**
- unsigned char **mapped**
- unsigned long **pixel**
- unsigned char **r**

The documentation for this struct was generated from the following file:

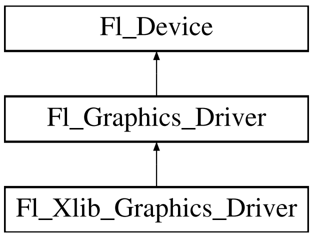
- Fl_XColor.H

31.157 Fl_Xlib_Graphics_Driver Class Reference

The Xlib-specific graphics class.

`#include <Fl_Device.H>`

Inheritance diagram for Fl_Xlib_Graphics_Driver:



Public Member Functions

- const char ∗ class_name ()

  Returns the name of the class of this object.
- void color (Fl_Color c)

  see fl_color(Fl_Color c).
- void color (uchar r, uchar g, uchar b)

  see fl_color(uchar r, uchar g, uchar b).
- void copy_offscreen (int x, int y, int w, int h, Fl_Offscreen pixmap, int srcx, int srcy)

  see fl_copy_offscreen()
- int descent ()

  see fl_descent().
- void draw (const char ∗str, int n, int x, int y)

  see fl_draw(const char ∗str, int n, int x, int y).
- void draw (int angle, const char ∗str, int n, int x, int y)

  see fl_draw(int angle, const char ∗str, int n, int x, int y).
- void draw (Fl_Pixmap ∗pxm, int XP, int YP, int WP, int HP, int cx, int cy)

  Draws an Fl_Pixmap object to the device.
- void draw (Fl_Bitmap ∗pxm, int XP, int YP, int WP, int HP, int cx, int cy)

  Draws an Fl_Bitmap object to the device.
- void draw (Fl_RGB_Image ∗img, int XP, int YP, int WP, int HP, int cx, int cy)

  Draws an Fl_RGB_Image object to the device.
- void draw_image (const uchar ∗buf, int X, int Y, int W, int H, int D=3, int L=0)

  see fl_draw_image(const uchar∗ buf, int X,int Y,int W,int H, int D, int L).
- void draw_image (Fl_Draw_Image_Cb cb, void ∗data, int X, int Y, int W, int H, int D=3)

  see fl_draw_image(Fl_Draw_Image_Cb cb, void∗ data, int X,int Y,int W,int H, int D).
- void draw_image_mono (const uchar ∗buf, int X, int Y, int W, int H, int D=1, int L=0)

  see fl_draw_image_mono(const uchar∗ buf, int X,int Y,int W,int H, int D, int L).
- void draw_image_mono (Fl_Draw_Image_Cb cb, void ∗data, int X, int Y, int W, int H, int D=1)

  see fl_draw_image_mono(Fl_Draw_Image_Cb cb, void∗ data, int X,int Y,int W,int H, int D).
- void font (Fl_Font face, Fl_Fontsize size)

  see fl_font(Fl_Font face, Fl_Fontsize size).
- int height ()

  see fl_height().
- void rtl_draw (const char ∗str, int n, int x, int y)

  see fl_rtl_draw(const char ∗str, int n, int x, int y).
- void text_extents (const char ∗, int n, int &dx, int &dy, int &w, int &h)

  see fl_text_extents(const char∗, int n, int& dx, int& dy, int& w, int& h).
- double width (const char ∗str, int n)

  see fl_width(const char ∗str, int n).
- double width (unsigned int c)

  see fl_width(unsigned int n).

Static Public Attributes

- static const char ∗ **class_id** = ”Fl_Xlib_Graphics_Driver”

Additional Inherited Members

31.157.1 Detailed Description

The Xlib-specific graphics class.

This class is implemented only on the Xlib platform.

31.157.2 Member Function Documentation

const char∗ Fl_Xlib_Graphics_Driver::class_name ( ) [inline], [virtual]

Returns the name of the class of this object.

Use of the class_name() function is discouraged because it will be removed from future FLTK versions.

The class of an instance of an Fl_Device subclass can be checked with code such as:

```
if ( instance->class_name() == Fl_Printer::class_id ) { ... }
```

Reimplemented from Fl_Graphics_Driver.

void Fl_Xlib_Graphics_Driver::color (Fl_Color *c*) [virtual]

see fl_color(Fl_Color c).

Reimplemented from Fl_Graphics_Driver.

void Fl_Xlib_Graphics_Driver::color (uchar *r*, uchar *g*, uchar *b*) [virtual]

see fl_color(uchar r, uchar g, uchar b).

Reimplemented from Fl_Graphics_Driver.

int Fl_Xlib_Graphics_Driver::descent ( ) [virtual]

see fl_descent().

Reimplemented from Fl_Graphics_Driver.

void Fl_Xlib_Graphics_Driver::draw (const char ∗ *str*, int *n*, int *x*, int *y*) [virtual]

see fl_draw(const char ∗str, int n, int x, int y).

Reimplemented from Fl_Graphics_Driver.

void Fl_Xlib_Graphics_Driver::draw (int *angle*, const char ∗ *str*, int *n*, int *x*, int *y*) [virtual]

see fl_draw(int angle, const char ∗str, int n, int x, int y).

Reimplemented from Fl_Graphics_Driver.

void Fl_Xlib_Graphics_Driver::draw (Fl_Pixmap ∗ *pxm*, int *XP*, int *YP*, int *WP*, int *HP*, int *cx*, int *cy*) [virtual]

Draws an Fl_Pixmap object to the device.

Specifies a bounding box for the image, with the origin (upper left-hand corner) of the image offset by the cx and cy arguments.

Reimplemented from Fl_Graphics_Driver.

void Fl_Xlib_Graphics_Driver::draw (Fl_Bitmap ∗ *bm*, int *XP*, int *YP*, int *WP*, int *HP*, int *cx*, int *cy*) `[virtual]`

Draws an Fl_Bitmap object to the device.

Specifies a bounding box for the image, with the origin (upper left-hand corner) of the image offset by the cx and cy arguments.

Reimplemented from Fl_Graphics_Driver.

void Fl_Xlib_Graphics_Driver::draw (Fl_RGB_Image ∗ *rgb*, int *XP*, int *YP*, int *WP*, int *HP*, int *cx*, int *cy*) `[virtual]`

Draws an Fl_RGB_Image object to the device.

Specifies a bounding box for the image, with the origin (upper left-hand corner) of the image offset by the cx and cy arguments.

Reimplemented from Fl_Graphics_Driver.

void Fl_Xlib_Graphics_Driver::draw_image (const uchar ∗ *buf*, int *X*, int *Y*, int *W*, int *H*, int *D* = *3*, int *L* = *0*) `[virtual]`

see fl_draw_image(const uchar∗ buf, int X,int Y,int W,int H, int D, int L).

Reimplemented from Fl_Graphics_Driver.

void Fl_Xlib_Graphics_Driver::draw_image (Fl_Draw_Image_Cb *cb*, void ∗ *data*, int *X*, int *Y*, int *W*, int *H*, int *D* = *3*) `[virtual]`

see fl_draw_image(Fl_Draw_Image_Cb cb, void∗ data, int X,int Y,int W,int H, int D).

Reimplemented from Fl_Graphics_Driver.

void Fl_Xlib_Graphics_Driver::draw_image_mono (const uchar ∗ *buf*, int *X*, int *Y*, int *W*, int *H*, int *D* = *1*, int *L* = *0*) `[virtual]`

see fl_draw_image_mono(const uchar∗ buf, int X,int Y,int W,int H, int D, int L).

Reimplemented from Fl_Graphics_Driver.

void Fl_Xlib_Graphics_Driver::draw_image_mono (Fl_Draw_Image_Cb *cb*, void ∗ *data*, int *X*, int *Y*, int *W*, int *H*, int *D* = *1*) `[virtual]`

see fl_draw_image_mono(Fl_Draw_Image_Cb cb, void∗ data, int X,int Y,int W,int H, int D).

Reimplemented from Fl_Graphics_Driver.

void Fl_Xlib_Graphics_Driver::font (Fl_Font *face*, Fl_Fontsize *fsize*) `[virtual]`

see fl_font(Fl_Font face, Fl_Fontsize size).

Reimplemented from Fl_Graphics_Driver.

int Fl_Xlib_Graphics_Driver::height ( ) `[virtual]`

see fl_height().

Reimplemented from Fl_Graphics_Driver.

void Fl_Xlib_Graphics_Driver::rtl_draw (const char ∗ *str*, int *n*, int *x*, int *y*) `[virtual]`

see fl_rtl_draw(const char ∗str, int n, int x, int y).

Reimplemented from Fl_Graphics_Driver.

void Fl_Xlib_Graphics_Driver::text_extents (const char * *t*, int *n*, int & *dx*, int & *dy*, int & *w*, int & *h*) `[virtual]`

see fl_text_extents(const char*, int n, int& dx, int& dy, int& w, int& h).

Reimplemented from Fl_Graphics_Driver.

double Fl_Xlib_Graphics_Driver::width (const char * *str*, int *n*) `[virtual]`

see fl_width(const char *str, int n).

Reimplemented from Fl_Graphics_Driver.

double Fl_Xlib_Graphics_Driver::width (unsigned int *c*) `[virtual]`

see fl_width(unsigned int n).

Reimplemented from Fl_Graphics_Driver.

The documentation for this class was generated from the following files:

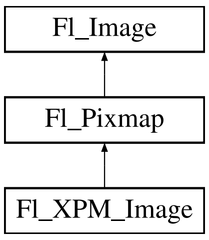
- Fl_Device.H
- Fl_Bitmap.cxx
- fl_color.cxx
- Fl_Device.cxx
- fl_draw_image.cxx
- Fl_Image.cxx
- Fl_Pixmap.cxx

31.158 Fl_XPM_Image Class Reference

The Fl_XPM_Image class supports loading, caching, and drawing of X Pixmap (XPM) images, including transparency.

`#include <Fl_XPM_Image.H>`

Inheritance diagram for Fl_XPM_Image:



Public Member Functions

- Fl_XPM_Image (const char *filename)

  The constructor loads the XPM image from the name filename.

Additional Inherited Members

31.158.1 Detailed Description

The Fl_XPM_Image class supports loading, caching, and drawing of X Pixmap (XPM) images, including transparency.

31.158.2 Constructor & Destructor Documentation

Fl_XPM_Image::Fl_XPM_Image (const char ∗ *name*)

The constructor loads the XPM image from the name filename.

The destructor frees all memory and server resources that are used by the image.

The documentation for this class was generated from the following files:

- Fl_XPM_Image.H
- Fl_XPM_Image.cxx

31.159 Fl_Text_Editor::Key_Binding Struct Reference

Simple linked list item associating a key/state to a function.

`#include <Fl_Text_Editor.H>`

Public Attributes

- Key_Func function

  associated function
- int key

  the key pressed
- Key_Binding ∗ next

  next key binding in the list
- int state

  the state of key modifiers

31.159.1 Detailed Description

Simple linked list item associating a key/state to a function.

The documentation for this struct was generated from the following file:

- Fl_Text_Editor.H

31.160 Fl_Graphics_Driver::matrix Struct Reference

A 2D coordinate transformation matrix.

`#include <Fl_Device.H>`

Public Attributes

- double **a**
- double **b**
- double **c**
- double **d**
- double **x**
- double **y**

31.160.1 Detailed Description

A 2D coordinate transformation matrix.

The documentation for this struct was generated from the following file:

- Fl_Device.H

31.161 Fl_Preferences::Name Class Reference

'Name' provides a simple method to create numerical or more complex procedural names for entries and groups on the fly.

```
#include <Fl_Preferences.H>
```

Public Member Functions

- Name (unsigned int n)

  Creates a group name or entry name on the fly.
- Name (const char *format,...)

  Creates a group name or entry name on the fly.
- operator const char * ()

  Return the Name as a "C" string.

31.161.1 Detailed Description

'Name' provides a simple method to create numerical or more complex procedural names for entries and groups on the fly.

Example: prefs.set(Fl_Preferences::Name("File%d",i),file[i]);.

See test/preferences.cxx as a sample for writing arrays into preferences.

'Name' is actually implemented as a class inside Fl_Preferences. It casts into const char* and gets automatically destroyed after the enclosing call ends.

31.161.2 Constructor & Destructor Documentation

Fl_Preferences::Name::Name (unsigned int *n*)

Creates a group name or entry name on the fly.

This version creates a simple unsigned integer as an entry name.

```
int n, i;
Fl_Preferences prev( appPrefs, "PreviousFiles" );
prev.get( "n", 0 );
for ( i=0; i<n; i++ )
  prev.get( Fl_Preferences::Name(i), prevFile[i], "" );
```

Fl_Preferences::Name::Name (const char * *format*, ...)

Creates a group name or entry name on the fly.

This version creates entry names as in 'printf'.

```
int n, i;
Fl_Preferences prefs( USER, "matthiasm.com", "test" );
prev.get( "nFiles", 0 );
for ( i=0; i<n; i++ )
  prev.get( Fl_Preferences::Name( "File%d", i ), prevFile[i], "" );
```

The documentation for this class was generated from the following files:

- Fl_Preferences.H
- Fl_Preferences.cxx

31.162 Fl_Preferences::Node Class Reference

Public Member Functions

- void **add** (const char ∗line)
- Node ∗ **addChild** (const char ∗path)
- const char ∗ **child** (int ix)
- Node ∗ **childNode** (int ix)
- void **deleteAllChildren** ()
- void **deleteAllEntries** ()
- char **deleteEntry** (const char ∗name)
- char **dirty** ()
- Entry & **entry** (int i)
- Node ∗ **find** (const char ∗path)
- RootNode ∗ **findRoot** ()
- const char ∗ **get** (const char ∗name)
- int **getEntry** (const char ∗name)
- const char ∗ **name** ()
- int **nChildren** ()
- int **nEntry** ()
- **Node** (const char ∗path)
- Node ∗ **parent** ()
- const char ∗ **path** ()
- char **remove** ()
- Node ∗ **search** (const char ∗path, int offset=0)
- void **set** (const char ∗name, const char ∗value)
- void **set** (const char ∗line)
- void **setParent** (Node ∗parent)
- void **setRoot** (RootNode ∗r)
- int **write** (FILE ∗f)

Static Public Attributes

- static int **lastEntrySet** = -1

The documentation for this class was generated from the following files:

- Fl_Preferences.H
- Fl_Preferences.cxx

31.163 Fl_Paged_Device::page_format Struct Reference

width, height and name of a page format

```
#include <Fl_Paged_Device.H>
```

Public Attributes

- int height

  height in points
- const char ∗ name

  format name
- int width

  width in points

31.163.1 Detailed Description

width, height and name of a page format

The documentation for this struct was generated from the following file:

- Fl_Paged_Device.H

31.164 Fl_Preferences::RootNode Class Reference

Public Member Functions

- char **getPath** (char ∗path, int pathlen)
- int **read** ()
- **RootNode** (Fl_Preferences ∗, Root root, const char ∗vendor, const char ∗application)
- **RootNode** (Fl_Preferences ∗, const char ∗path, const char ∗vendor, const char ∗application)
- **RootNode** (Fl_Preferences ∗)
- int **write** ()

The documentation for this class was generated from the following files:

- Fl_Preferences.H
- Fl_Preferences.cxx

31.165 Fl_Window::shape_data_type Struct Reference

Data supporting a non-rectangular window shape.

`#include <Fl_Window.H>`

Public Attributes

- int lh_

  height of shape image
- int lw_

  width of shape image
- Fl_Image ∗ shape_

  shape image
- Fl_Bitmap ∗ todelete_

  auxiliary bitmap image

31.165.1 Detailed Description

Data supporting a non-rectangular window shape.

The documentation for this struct was generated from the following file:

- Fl_Window.H

31.166 Fl_Text_Display::Style_Table_Entry Struct Reference

This structure associates the color, font, and font size of a string to draw with an attribute mask matching attr.

`#include <Fl_Text_Display.H>`

Public Attributes

- unsigned attr

  currently unused (this may be change in the future)
- Fl_Color color

  text color
- Fl_Font font

  text font
- Fl_Fontsize size

  text font size

31.166.1 Detailed Description

This structure associates the color, font, and font size of a string to draw with an attribute mask matching attr.

There must be one entry for each style that can be used in an Fl_Text_Display for displaying text. The style table is an array of struct Style_Table_Entry.

The style table is associated with an Fl_Text_Display by using Fl_Text_Display::highlight_data().

See Also

Fl_Text_Display::highlight_data()

The documentation for this struct was generated from the following file:

- Fl_Text_Display.H

Windows/Linux	**Mac**	**Function**
^A	**Command-A**	**Selects all text in the widget.**
^C	**Command-C**	**Copy the current selection to the clipboard.**
^I	**^I**	**Insert a tab.**
^J	**^J**	**Insert a Line Feed.** (Similar to literal ’Enter’ character)
^L	**^L**	**Insert a Form Feed.**
^M	**^M**	**Insert a Carriage Return.**
^V, **Shift-Insert**	**Command-V**	**Paste the clipboard.** (Macs keyboards don’t have ”Insert” keys, but if they did, Shift-Insert would work)
^X, **Shift-Delete**	**Command-X,** **Shift-Delete**	**Cut.** Copy the selection to the clipboard and delete it. (If there’s no selection, Shift-Delete acts like Delete)
^Z	**Command-Z**	**Undo.** This is a single-level undo mechanism, but all adjacent deletions and insertions are concatenated into a single ”undo”. Often this will undo a lot more than you expected.
Shift-^Z	**Shift-Command-Z**	**Redo.** Currently same behavior as ^Z. Reserved for future multilevel undo/redo.
Arrow Keys	**Arrow Keys**	**Standard cursor movement.** Can be combined with Shift to extend selection.
Home	**Command-Up,** **Command-Left**	**Move to start of line.** Can be combined with Shift to extend selection.
End	**Command-Down,** **Command-Right**	**Move to end of line.** Can be combined with Shift to extend selection.
Ctrl-Home	**Command-Up,** **Command-PgUp,** **Ctrl-Left**	**Move to top of document/field.** In single line input, moves to start of line. In multiline input, moves to start of top line. Can be combined with Shift to extend selection.
Ctrl-End	**Command-End,**	**Move to bottom of**

Keyboard	**FL_TREE_SELECT_-MULTI**	**FL_TREE_SELECT_-SINGLE**	**FL_TREE_SELECT_-NONE**
Ctrl-A (Linux/Windows) **Command-A** (Mac)	Select all items.	N/A	N/A
Space	Selects item.	Selects item.	N/A
Ctrl-Space	Toggle item.	Toggle item.	N/A
Shift-Space	Extends selection from last item.	Selects item.	N/A
Enter, Ctrl-Enter, Shift-Enter	Toggles open/close	Toggles open/close	Toggles open/close
Right / Left	Open/Close item.	Open/Close item.	Open/Close item.
Up / Down	Move focus box up/down.	Move focus box up/down.	N/A
Shift-Up / Shift-Down	Extend selection up/down.	Move focus up/down.	N/A
Home / End	Move to top/bottom of tree.	Move to top/bottom of tree.	Move to top/bottom of tree.
PageUp / PageDown	Page up/down.	Page up/down.	Page up/down.

Table 31.3: Fl_Tree keyboard bindings.

Chapter 32

File Documentation

32.1 Enumerations.H File Reference

This file contains type definitions and general enumerations.

```
   #include <FL/abi-version.h>
#include "Fl_Export.H"
#include "fl_types.h"
```

Macros

- #define **FL_IMAGE_WITH_ALPHA** 0x40000000

Version Numbers

FLTK defines some constants to help the programmer to find out, for which FLTK version a program is compiled.

The following constants are defined:

- #define FL_MAJOR_VERSION 1

  The major release version of this FLTK library.
- #define FL_MINOR_VERSION 3

  The minor release version for this library.
- #define FL_PATCH_VERSION 4

  The patch version for this library.
- #define FL_VERSION

  The FLTK version number as a double.
- #define FL_API_VERSION (FL_MAJOR_VERSION∗10000 + FL_MINOR_VERSION∗100 + F-L_PATCH_VERSION)

  The FLTK API version number as an int.
- #define FL_ABI_VERSION (FL_MAJOR_VERSION∗10000 + FL_MINOR_VERSION∗100)

  The FLTK ABI (Application Binary Interface) version number as an int.
- #define **FLTK_ABI_VERSION** FL_ABI_VERSION

Mouse and Keyboard Events

```
This and the following constants define the non-ASCII keys on the
keyboard for FL_KEYBOARD and FL_SHORTCUT events.
```

Todo *FL_Button and FL_key... constants could be structured better (use an enum or some doxygen grouping ?)*

```
\sa Fl::event_key() and Fl::get_key(int) (use ascii letters for all other keys):
```

- #define FL_Button 0xfee8

  A mouse button; use Fl_Button + n for mouse button n.
- #define FL_BackSpace 0xff08

  The backspace key.
- #define FL_Tab 0xff09

  The tab key.
- #define FL_Iso_Key 0xff0c

  The additional key of ISO keyboards.
- #define FL_Enter 0xff0d

  The enter key.
- #define FL_Pause 0xff13

  The pause key.
- #define FL_Scroll_Lock 0xff14

  The scroll lock key.
- #define FL_Escape 0xff1b

  The escape key.
- #define FL_Kana 0xff2e

  The Kana key of JIS keyboards.
- #define FL_Eisu 0xff2f

  The Eisu key of JIS keyboards.
- #define FL_Yen 0xff30

  The Yen key of JIS keyboards.
- #define FL_JIS_Underscore 0xff31

  The underscore key of JIS keyboards.
- #define FL_Home 0xff50

  The home key.
- #define FL_Left 0xff51

  The left arrow key.
- #define FL_Up 0xff52

  The up arrow key.
- #define FL_Right 0xff53

  The right arrow key.
- #define FL_Down 0xff54

  The down arrow key.
- #define FL_Page_Up 0xff55

  The page-up key.
- #define FL_Page_Down 0xff56

  The page-down key.
- #define FL_End 0xff57

  The end key.
- #define FL_Print 0xff61

  The print (or print-screen) key.
- #define FL_Insert 0xff63

  The insert key.
- #define FL_Menu 0xff67

  The menu key.
- #define FL_Help 0xff68

  The 'help' key on Mac keyboards.
- #define FL_Num_Lock 0xff7f

The num lock key.

- #define FL_KP 0xff80

  One of the keypad numbers; use FL_KP + 'n' for digit n.
- #define FL_KP_Enter 0xff8d

  The enter key on the keypad, same as Fl_KP+'\r'.
- #define FL_KP_Last 0xffbd

  The last keypad key; use to range-check keypad.
- #define FL_F 0xffbd

  One of the function keys; use FL_F + n for function key n.
- #define FL_F_Last 0xffe0

  The last function key; use to range-check function keys.
- #define FL_Shift_L 0xffe1

  The lefthand shift key.
- #define FL_Shift_R 0xffe2

  The righthand shift key.
- #define FL_Control_L 0xffe3

  The lefthand control key.
- #define FL_Control_R 0xffe4

  The righthand control key.
- #define FL_Caps_Lock 0xffe5

  The caps lock key.
- #define FL_Meta_L 0xffe7

  The left meta/Windows key.
- #define FL_Meta_R 0xffe8

  The right meta/Windows key.
- #define FL_Alt_L 0xffe9

  The left alt key.
- #define FL_Alt_R 0xffea

  The right alt key.
- #define FL_Delete 0xffff

  The delete key.
- #define **FL_Volume_Down** 0xEF11 /∗ Volume control down ∗/
- #define **FL_Volume_Mute** 0xEF12 /∗ Mute sound from the system ∗/
- #define **FL_Volume_Up** 0xEF13 /∗ Volume control up ∗/
- #define **FL_Media_Play** 0xEF14 /∗ Start playing of audio ∗/
- #define **FL_Media_Stop** 0xEF15 /∗ Stop playing audio ∗/
- #define **FL_Media_Prev** 0xEF16 /∗ Previous track ∗/
- #define **FL_Media_Next** 0xEF17 /∗ Next track ∗/
- #define **FL_Home_Page** 0xEF18 /∗ Display user's home page ∗/
- #define **FL_Mail** 0xEF19 /∗ Invoke user's mail program ∗/
- #define **FL_Search** 0xEF1B /∗ Search ∗/
- #define **FL_Back** 0xEF26 /∗ Like back on a browser ∗/
- #define **FL_Forward** 0xEF27 /∗ Like forward on a browser ∗/
- #define **FL_Stop** 0xEF28 /∗ Stop current operation ∗/
- #define **FL_Refresh** 0xEF29 /∗ Refresh the page ∗/
- #define **FL_Sleep** 0xEF2F /∗ Put system to sleep ∗/
- #define **FL_Favorites** 0xEF30 /∗ Show favorite locations ∗/

Mouse Buttons

```
These constants define the button numbers for FL_PUSH and FL_RELEASE events.
```

See Also

Fl::event_button()

- #define FL_LEFT_MOUSE 1

  The left mouse button.
- #define FL_MIDDLE_MOUSE 2

  The middle mouse button.
- #define FL_RIGHT_MOUSE 3

  The right mouse button.

Event States

```
The following constants define bits in the Fl::event_state() value.
```

- #define FL_SHIFT 0x00010000

  One of the shift keys is down.
- #define FL_CAPS_LOCK 0x00020000

  The caps lock is on.
- #define FL_CTRL 0x00040000

  One of the ctrl keys is down.
- #define FL_ALT 0x00080000

  One of the alt keys is down.
- #define FL_NUM_LOCK 0x00100000

  The num lock is on.
- #define FL_META 0x00400000

  One of the meta/Windows keys is down.
- #define FL_SCROLL_LOCK 0x00800000

  The scroll lock is on.
- #define FL_BUTTON1 0x01000000

  Mouse button 1 is pushed.
- #define FL_BUTTON2 0x02000000

  Mouse button 2 is pushed.
- #define FL_BUTTON3 0x04000000

  Mouse button 3 is pushed.
- #define FL_BUTTONS 0x7f000000

  Any mouse button is pushed.
- #define FL_BUTTON(n) (0x00800000<<(n))

  Mouse button n (n > 0) is pushed.
- #define FL_KEY_MASK 0x0000ffff

  All keys are 16 bit for now.
- #define FL_COMMAND FL_CTRL

  An alias for FL_CTRL on WIN32 and X11, or FL_META on MacOS X.
- #define FL_CONTROL FL_META

  An alias for FL_META on WIN32 and X11, or FL_CTRL on MacOS X.

Typedefs

- typedef int Fl_Fontsize

  Size of a font in pixels.

Enumerations

- enum { FL_READ = 1, FL_WRITE = 4, FL_EXCEPT = 8 }

  FD "when" conditions.
- enum Fl_Damage {
  FL_DAMAGE_CHILD = 0x01, FL_DAMAGE_EXPOSE = 0x02, FL_DAMAGE_SCROLL = 0x04, FL_DAMAGE_OVERLAY = 0x08,
  FL_DAMAGE_USER1 = 0x10, FL_DAMAGE_USER2 = 0x20, FL_DAMAGE_ALL = 0x80 }

  Damage masks.
- enum Fl_Event {
  FL_NO_EVENT = 0, FL_PUSH = 1, FL_RELEASE = 2, FL_ENTER = 3,
  FL_LEAVE = 4, FL_DRAG = 5, FL_FOCUS = 6, FL_UNFOCUS = 7,
  FL_KEYDOWN = 8, FL_KEYBOARD = 8, FL_KEYUP = 9, FL_CLOSE = 10,
  FL_MOVE = 11, FL_SHORTCUT = 12, FL_DEACTIVATE = 13, FL_ACTIVATE = 14,
  FL_HIDE = 15, FL_SHOW = 16, FL_PASTE = 17, FL_SELECTIONCLEAR = 18,
  FL_MOUSEWHEEL = 19, FL_DND_ENTER = 20, FL_DND_DRAG = 21, FL_DND_LEAVE = 22,
  FL_DND_RELEASE = 23, FL_SCREEN_CONFIGURATION_CHANGED = 24, FL_FULLSCRE-EN = 25, FL_ZOOM_GESTURE = 26 }

  Every time a user moves the mouse pointer, clicks a button, or presses a key, an event is generated and sent to your application.
- enum Fl_Labeltype {
  FL_NORMAL_LABEL = 0, FL_NO_LABEL, _FL_SHADOW_LABEL, _FL_ENGRAVED_LABEL,
  _FL_EMBOSSED_LABEL, _FL_MULTI_LABEL, _FL_ICON_LABEL, _FL_IMAGE_LABEL,
  FL_FREE_LABELTYPE }

  The labeltype() method sets the type of the label.
- enum Fl_Mode {
  FL_RGB = 0, **FL_INDEX** = 1, **FL_SINGLE** = 0, **FL_DOUBLE** = 2,
  FL_ACCUM = 4, **FL_ALPHA** = 8, **FL_DEPTH** = 16, **FL_STENCIL** = 32,
  FL_RGB8 = 64, **FL_MULTISAMPLE** = 128, **FL_STEREO** = 256, **FL_FAKE_SINGLE** = 512,
  FL_OPENGL3 = 1024 }

  visual types and Fl_Gl_Window::mode() (values match Glut)

When Conditions

- enum Fl_When {
  FL_WHEN_NEVER = 0, FL_WHEN_CHANGED = 1, FL_WHEN_NOT_CHANGED = 2, FL_-WHEN_RELEASE = 4,
  FL_WHEN_RELEASE_ALWAYS = 6, FL_WHEN_ENTER_KEY = 8, FL_WHEN_ENTER_KE-Y_ALWAYS =10, FL_WHEN_ENTER_KEY_CHANGED =11 }

  These constants determine when a callback is performed.

Cursors

- enum Fl_Cursor {
  FL_CURSOR_DEFAULT = 0, FL_CURSOR_ARROW = 35, FL_CURSOR_CROSS = 66, FL_C-URSOR_WAIT = 76,
  FL_CURSOR_INSERT = 77, FL_CURSOR_HAND = 31, FL_CURSOR_HELP = 47, FL_CURS-OR_MOVE = 27,
  FL_CURSOR_NS = 78, FL_CURSOR_WE = 79, FL_CURSOR_NWSE = 80, FL_CURSOR_NE-SW = 81,
  FL_CURSOR_N = 70, FL_CURSOR_NE = 69, FL_CURSOR_E = 49, FL_CURSOR_SE = 8,
  FL_CURSOR_S = 9, FL_CURSOR_SW = 7, FL_CURSOR_W = 36, FL_CURSOR_NW = 68,
  FL_CURSOR_NONE =255 }

  The following constants define the mouse cursors that are available in FLTK.

Variables

- FL_EXPORT Fl_Fontsize FL_NORMAL_SIZE

  normal font size

Box Types

FLTK standard box types

This enum defines the standard box types included with FLTK.

FL_NO_BOX means nothing is drawn at all, so whatever is already on the screen remains. The FL_..._-FRAME types only draw their edges, leaving the interior unchanged. The blue color in Figure 1 is the area that is not drawn by the frame types.

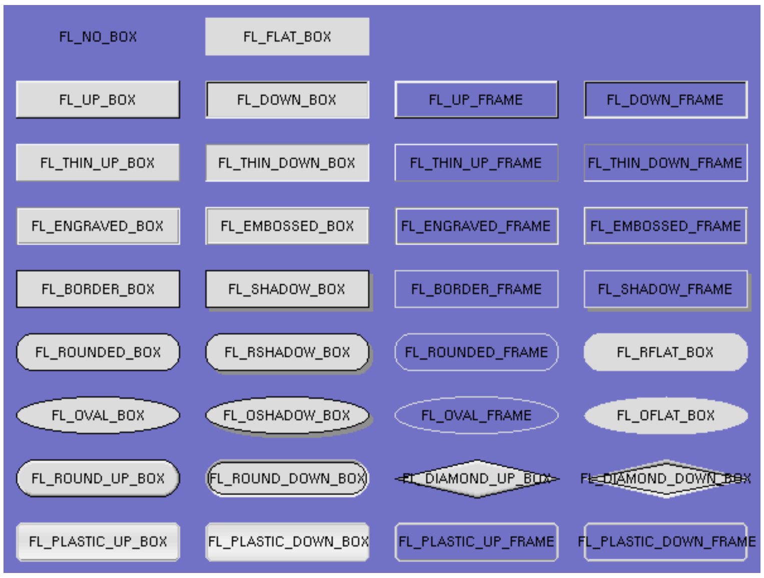


Figure 32.1: FLTK standard box types

Todo Description of boxtypes is incomplete. See below for the defined enum Fl_Boxtype.

See Also

src/Fl_get_system_colors.cxx

- #define **FL_ROUND_UP_BOX** fl_define_FL_ROUND_UP_BOX()
- #define **FL_ROUND_DOWN_BOX** (Fl_Boxtype)(fl_define_FL_ROUND_UP_BOX()+1)
- #define **FL_SHADOW_BOX** fl_define_FL_SHADOW_BOX()
- #define **FL_SHADOW_FRAME** (Fl_Boxtype)(fl_define_FL_SHADOW_BOX()+2)
- #define **FL_ROUNDED_BOX** fl_define_FL_ROUNDED_BOX()
- #define **FL_ROUNDED_FRAME** (Fl_Boxtype)(fl_define_FL_ROUNDED_BOX()+2)
- #define **FL_RFLAT_BOX** fl_define_FL_RFLAT_BOX()
- #define **FL_RSHADOW_BOX** fl_define_FL_RSHADOW_BOX()
- #define **FL_DIAMOND_UP_BOX** fl_define_FL_DIAMOND_BOX()
- #define **FL_DIAMOND_DOWN_BOX** (Fl_Boxtype)(fl_define_FL_DIAMOND_BOX()+1)
- #define **FL_OVAL_BOX** fl_define_FL_OVAL_BOX()
- #define **FL_OSHADOW_BOX** (Fl_Boxtype)(fl_define_FL_OVAL_BOX()+1)
- #define **FL_OVAL_FRAME** (Fl_Boxtype)(fl_define_FL_OVAL_BOX()+2)

- #define **FL_OFLAT_BOX** (Fl_Boxtype)(fl_define_FL_OVAL_BOX()+3)
- #define **FL_PLASTIC_UP_BOX** fl_define_FL_PLASTIC_UP_BOX()
- #define **FL_PLASTIC_DOWN_BOX** (Fl_Boxtype)(fl_define_FL_PLASTIC_UP_BOX()+1)
- #define **FL_PLASTIC_UP_FRAME** (Fl_Boxtype)(fl_define_FL_PLASTIC_UP_BOX()+2)
- #define **FL_PLASTIC_DOWN_FRAME** (Fl_Boxtype)(fl_define_FL_PLASTIC_UP_BOX()+3)
- #define **FL_PLASTIC_THIN_UP_BOX** (Fl_Boxtype)(fl_define_FL_PLASTIC_UP_BOX()+4)
- #define **FL_PLASTIC_THIN_DOWN_BOX** (Fl_Boxtype)(fl_define_FL_PLASTIC_UP_BOX()+5)
- #define **FL_PLASTIC_ROUND_UP_BOX** (Fl_Boxtype)(fl_define_FL_PLASTIC_UP_BOX()+6)
- #define **FL_PLASTIC_ROUND_DOWN_BOX** (Fl_Boxtype)(fl_define_FL_PLASTIC_UP_BOX()+7)
- #define **FL_GTK_UP_BOX** fl_define_FL_GTK_UP_BOX()
- #define **FL_GTK_DOWN_BOX** (Fl_Boxtype)(fl_define_FL_GTK_UP_BOX()+1)
- #define **FL_GTK_UP_FRAME** (Fl_Boxtype)(fl_define_FL_GTK_UP_BOX()+2)
- #define **FL_GTK_DOWN_FRAME** (Fl_Boxtype)(fl_define_FL_GTK_UP_BOX()+3)
- #define **FL_GTK_THIN_UP_BOX** (Fl_Boxtype)(fl_define_FL_GTK_UP_BOX()+4)
- #define **FL_GTK_THIN_DOWN_BOX** (Fl_Boxtype)(fl_define_FL_GTK_UP_BOX()+5)
- #define **FL_GTK_THIN_UP_FRAME** (Fl_Boxtype)(fl_define_FL_GTK_UP_BOX()+6)
- #define **FL_GTK_THIN_DOWN_FRAME** (Fl_Boxtype)(fl_define_FL_GTK_UP_BOX()+7)
- #define **FL_GTK_ROUND_UP_BOX** (Fl_Boxtype)(fl_define_FL_GTK_UP_BOX()+8)
- #define **FL_GTK_ROUND_DOWN_BOX** (Fl_Boxtype)(fl_define_FL_GTK_UP_BOX()+9)
- #define **FL_GLEAM_UP_BOX** fl_define_FL_GLEAM_UP_BOX()
- #define **FL_GLEAM_DOWN_BOX** (Fl_Boxtype)(fl_define_FL_GLEAM_UP_BOX()+1)
- #define **FL_GLEAM_UP_FRAME** (Fl_Boxtype)(fl_define_FL_GLEAM_UP_BOX()+2)
- #define **FL_GLEAM_DOWN_FRAME** (Fl_Boxtype)(fl_define_FL_GLEAM_UP_BOX()+3)
- #define **FL_GLEAM_THIN_UP_BOX** (Fl_Boxtype)(fl_define_FL_GLEAM_UP_BOX()+4)
- #define **FL_GLEAM_THIN_DOWN_BOX** (Fl_Boxtype)(fl_define_FL_GLEAM_UP_BOX()+5)
- #define **FL_GLEAM_ROUND_UP_BOX** (Fl_Boxtype)(fl_define_FL_GLEAM_UP_BOX()+6)
- #define **FL_GLEAM_ROUND_DOWN_BOX** (Fl_Boxtype)(fl_define_FL_GLEAM_UP_BOX()+7)
- #define **FL_FRAME** FL_ENGRAVED_FRAME
- #define **FL_FRAME_BOX** FL_ENGRAVED_BOX
- #define **FL_CIRCLE_BOX** FL_ROUND_DOWN_BOX
- #define **FL_DIAMOND_BOX** FL_DIAMOND_DOWN_BOX
- enum Fl_Boxtype {
  FL_NO_BOX = 0, FL_FLAT_BOX, FL_UP_BOX, FL_DOWN_BOX,
  FL_UP_FRAME, FL_DOWN_FRAME, FL_THIN_UP_BOX, FL_THIN_DOWN_BOX,
  FL_THIN_UP_FRAME, FL_THIN_DOWN_FRAME, FL_ENGRAVED_BOX, FL_EMBOSSED_B-
  OX,
  FL_ENGRAVED_FRAME, FL_EMBOSSED_FRAME, FL_BORDER_BOX, _FL_SHADOW_BOX,
  FL_BORDER_FRAME, _FL_SHADOW_FRAME, _FL_ROUNDED_BOX, _FL_RSHADOW_BOX,
  _FL_ROUNDED_FRAME, _FL_RFLAT_BOX, _FL_ROUND_UP_BOX, _FL_ROUND_DOWN_B-
  OX,
  _FL_DIAMOND_UP_BOX, _FL_DIAMOND_DOWN_BOX, _FL_OVAL_BOX, _FL_OSHADOW_-
  BOX,
  _FL_OVAL_FRAME, _FL_OFLAT_BOX, _FL_PLASTIC_UP_BOX, _FL_PLASTIC_DOWN_BOX,
  _FL_PLASTIC_UP_FRAME, _FL_PLASTIC_DOWN_FRAME, _FL_PLASTIC_THIN_UP_BOX, _-
  FL_PLASTIC_THIN_DOWN_BOX,
  _FL_PLASTIC_ROUND_UP_BOX, _FL_PLASTIC_ROUND_DOWN_BOX, _FL_GTK_UP_BOX, _-
  FL_GTK_DOWN_BOX,
  _FL_GTK_UP_FRAME, _FL_GTK_DOWN_FRAME, _FL_GTK_THIN_UP_BOX, _FL_GTK_THIN-
  _DOWN_BOX,
  _FL_GTK_THIN_UP_FRAME, _FL_GTK_THIN_DOWN_FRAME, _FL_GTK_ROUND_UP_BOX, -
  _FL_GTK_ROUND_DOWN_BOX,
  _FL_GLEAM_UP_BOX, _FL_GLEAM_DOWN_BOX, _FL_GLEAM_UP_FRAME, _FL_GLEAM_-

DOWN_FRAME,
_FL_GLEAM_THIN_UP_BOX, _FL_GLEAM_THIN_DOWN_BOX, _FL_GLEAM_ROUND_UP_BOX, _FL_GLEAM_ROUND_DOWN_BOX,
FL_FREE_BOXTYPE }

- FL_EXPORT Fl_Boxtype **fl_define_FL_ROUND_UP_BOX** ()
- FL_EXPORT Fl_Boxtype **fl_define_FL_SHADOW_BOX** ()
- FL_EXPORT Fl_Boxtype **fl_define_FL_ROUNDED_BOX** ()
- FL_EXPORT Fl_Boxtype **fl_define_FL_RFLAT_BOX** ()
- FL_EXPORT Fl_Boxtype **fl_define_FL_RSHADOW_BOX** ()
- FL_EXPORT Fl_Boxtype **fl_define_FL_DIAMOND_BOX** ()
- FL_EXPORT Fl_Boxtype **fl_define_FL_OVAL_BOX** ()
- FL_EXPORT Fl_Boxtype **fl_define_FL_PLASTIC_UP_BOX** ()
- FL_EXPORT Fl_Boxtype **fl_define_FL_GTK_UP_BOX** ()
- FL_EXPORT Fl_Boxtype **fl_define_FL_GLEAM_UP_BOX** ()
- Fl_Boxtype fl_box (Fl_Boxtype b)

  Get the filled version of a frame.
- Fl_Boxtype fl_down (Fl_Boxtype b)

  Get the "pressed" or "down" version of a box.
- Fl_Boxtype fl_frame (Fl_Boxtype b)

  Get the unfilled, frame only version of a box.

- #define FL_SYMBOL_LABEL FL_NORMAL_LABEL

  Sets the current label type and return its corresponding Fl_Labeltype value.
- #define **FL_SHADOW_LABEL** fl_define_FL_SHADOW_LABEL()
- #define **FL_ENGRAVED_LABEL** fl_define_FL_ENGRAVED_LABEL()
- #define **FL_EMBOSSED_LABEL** fl_define_FL_EMBOSSED_LABEL()
- Fl_Labeltype FL_EXPORT **fl_define_FL_SHADOW_LABEL** ()
- Fl_Labeltype FL_EXPORT **fl_define_FL_ENGRAVED_LABEL** ()
- Fl_Labeltype FL_EXPORT **fl_define_FL_EMBOSSED_LABEL** ()

Alignment Flags

Flags to control the label alignment.

This controls how the label is displayed next to or inside the widget. The default value is FL_ALIGN_CENTER (0) for most widgets, which centers the label inside the widget.

Flags can be or'd to achieve a combination of alignments, but there are some "magic values" (e.g. combinations of TOP and BOTTOM and of LEFT and RIGHT) that have special meanings (see below). For instance:

FL_ALIGN_TOP_LEFT == (FL_ALIGN_TOP|FL_ALIGN_LEFT) != FL_ALIGN_LEFT_TOP.

```
Outside alignments (FL_ALIGN_INSIDE is not set):

           TOP_LEFT        TOP       TOP_RIGHT
           +---------------------------------+
   LEFT_TOP|                                 |RIGHT_TOP
           |                                 |
       LEFT|             CENTER              |RIGHT
           |                                 |
LEFT_BOTTOM|                                 |RIGHT_BOTTOM
           +---------------------------------+
           BOTTOM_LEFT   BOTTOM   BOTTOM_RIGHT

Inside alignments (FL_ALIGN_INSIDE is set):

           +---------------------------------+
           |TOP_LEFT        TOP     TOP_RIGHT|
           |                                 |
           |LEFT          CENTER        RIGHT|
           |                                 |
           |BOTTOM_LEFT   BOTTOM BOTTOM_RIGHT|
           +---------------------------------+
```

See Also

FL_ALIGN_CENTER, etc.

- typedef unsigned Fl_Align

  FLTK type for alignment control.
- const Fl_Align FL_ALIGN_CENTER = (Fl_Align)0

  Align the label horizontally in the middle.
- const Fl_Align FL_ALIGN_TOP = (Fl_Align)1

  Align the label at the top of the widget.
- const Fl_Align FL_ALIGN_BOTTOM = (Fl_Align)2

  Align the label at the bottom of the widget.
- const Fl_Align FL_ALIGN_LEFT = (Fl_Align)4

  Align the label at the left of the widget.
- const Fl_Align FL_ALIGN_RIGHT = (Fl_Align)8

  Align the label to the right of the widget.
- const Fl_Align FL_ALIGN_INSIDE = (Fl_Align)16

  Draw the label inside of the widget.
- const Fl_Align FL_ALIGN_TEXT_OVER_IMAGE = (Fl_Align)0x0020

  If the label contains an image, draw the text on top of the image.
- const Fl_Align FL_ALIGN_IMAGE_OVER_TEXT = (Fl_Align)0x0000

  If the label contains an image, draw the text below the image.
- const Fl_Align FL_ALIGN_CLIP = (Fl_Align)64

  All parts of the label that are lager than the widget will not be drawn .
- const Fl_Align FL_ALIGN_WRAP = (Fl_Align)128

  Wrap text that does not fit the width of the widget.
- const Fl_Align FL_ALIGN_IMAGE_NEXT_TO_TEXT = (Fl_Align)0x0100

  If the label contains an image, draw the text to the right of the image.
- const Fl_Align FL_ALIGN_TEXT_NEXT_TO_IMAGE = (Fl_Align)0x0120

  If the label contains an image, draw the text to the left of the image.
- const Fl_Align FL_ALIGN_IMAGE_BACKDROP = (Fl_Align)0x0200

  If the label contains an image, draw the image or deimage in the background.
- const Fl_Align **FL_ALIGN_TOP_LEFT** = FL_ALIGN_TOP | FL_ALIGN_LEFT
- const Fl_Align **FL_ALIGN_TOP_RIGHT** = FL_ALIGN_TOP | FL_ALIGN_RIGHT
- const Fl_Align **FL_ALIGN_BOTTOM_LEFT** = FL_ALIGN_BOTTOM | FL_ALIGN_LEFT
- const Fl_Align **FL_ALIGN_BOTTOM_RIGHT** = FL_ALIGN_BOTTOM | FL_ALIGN_RIGHT
- const Fl_Align **FL_ALIGN_LEFT_TOP** = 0x0007
- const Fl_Align **FL_ALIGN_RIGHT_TOP** = 0x000b
- const Fl_Align **FL_ALIGN_LEFT_BOTTOM** = 0x000d
- const Fl_Align **FL_ALIGN_RIGHT_BOTTOM** = 0x000e
- const Fl_Align **FL_ALIGN_NOWRAP** = (Fl_Align)0
- const Fl_Align **FL_ALIGN_POSITION_MASK** = 0x000f
- const Fl_Align **FL_ALIGN_IMAGE_MASK** = 0x0320

Font Numbers

The following constants define the standard FLTK fonts:

- typedef int Fl_Font

  A font number is an index into the internal font table.
- const Fl_Font FL_HELVETICA = 0

  Helvetica (or Arial) normal (0)
- const Fl_Font FL_HELVETICA_BOLD = 1

  Helvetica (or Arial) bold.
- const Fl_Font FL_HELVETICA_ITALIC = 2

  Helvetica (or Arial) oblique.
- const Fl_Font FL_HELVETICA_BOLD_ITALIC = 3

  Helvetica (or Arial) bold-oblique.
- const Fl_Font FL_COURIER = 4

  Courier normal.
- const Fl_Font FL_COURIER_BOLD = 5

  Courier bold.
- const Fl_Font FL_COURIER_ITALIC = 6

  Courier italic.
- const Fl_Font FL_COURIER_BOLD_ITALIC = 7

  Courier bold-italic.
- const Fl_Font FL_TIMES = 8

  Times roman.
- const Fl_Font FL_TIMES_BOLD = 9

  Times roman bold.
- const Fl_Font FL_TIMES_ITALIC = 10

  Times roman italic.
- const Fl_Font FL_TIMES_BOLD_ITALIC = 11

  Times roman bold-italic.
- const Fl_Font FL_SYMBOL = 12

  Standard symbol font.
- const Fl_Font FL_SCREEN = 13

  Default monospaced screen font.
- const Fl_Font FL_SCREEN_BOLD = 14

  Default monospaced bold screen font.
- const Fl_Font FL_ZAPF_DINGBATS = 15

  Zapf-dingbats font.
- const Fl_Font FL_FREE_FONT = 16

  first one to allocate
- const Fl_Font FL_BOLD = 1

  add this to helvetica, courier, or times
- const Fl_Font FL_ITALIC = 2

  add this to helvetica, courier, or times
- const Fl_Font FL_BOLD_ITALIC = 3

  add this to helvetica, courier, or times

Colors

The Fl_Color type holds an FLTK color value.

Colors are either 8-bit indexes into a `virtual colormap` or 24-bit RGB color values. (See Colors for the default FLTK colormap)

Color indices occupy the lower 8 bits of the value, while RGB colors occupy the upper 24 bits, for a byte organization of RGBI.

```
 Fl_Color => 0xrrggbbii
                | | | |
                | | | +--- index between 0 and 255
                | | +----- blue color component (8 bit)
                | +------- green component (8 bit)
                +--------- red component (8 bit)
```

A color can have either an index or an rgb value. Colors with rgb set and an index >0 are reserved for special use.

- #define **FL_FREE_COLOR** (Fl_Color)16
- #define **FL_NUM_FREE_COLOR** 16
- #define **FL_GRAY_RAMP** (Fl_Color)32
- #define **FL_NUM_GRAY** 24
- #define **FL_GRAY** FL_BACKGROUND_COLOR
- #define **FL_COLOR_CUBE** (Fl_Color)56
- #define **FL_NUM_RED** 5
- #define **FL_NUM_GREEN** 8
- #define **FL_NUM_BLUE** 5
- typedef unsigned int Fl_Color

  An FLTK color value; see also Colors.
- const Fl_Color FL_FOREGROUND_COLOR = 0

  the default foreground color (0) used for labels and text
- const Fl_Color FL_BACKGROUND2_COLOR = 7

  the default background color for text, list, and valuator widgets
- const Fl_Color FL_INACTIVE_COLOR = 8

  the inactive foreground color
- const Fl_Color FL_SELECTION_COLOR = 15

  the default selection/highlight color
- const Fl_Color **FL_GRAY0** = 32
- const Fl_Color **FL_DARK3** = 39
- const Fl_Color **FL_DARK2** = 45
- const Fl_Color **FL_DARK1** = 47
- const Fl_Color **FL_BACKGROUND_COLOR** = 49
- const Fl_Color **FL_LIGHT1** = 50
- const Fl_Color **FL_LIGHT2** = 52
- const Fl_Color **FL_LIGHT3** = 54
- const Fl_Color **FL_BLACK** = 56
- const Fl_Color **FL_RED** = 88
- const Fl_Color **FL_GREEN** = 63
- const Fl_Color **FL_YELLOW** = 95
- const Fl_Color **FL_BLUE** = 216
- const Fl_Color **FL_MAGENTA** = 248

- const Fl_Color **FL_CYAN** = 223
- const Fl_Color **FL_DARK_RED** = 72
- const Fl_Color **FL_DARK_GREEN** = 60
- const Fl_Color **FL_DARK_YELLOW** = 76
- const Fl_Color **FL_DARK_BLUE** = 136
- const Fl_Color **FL_DARK_MAGENTA** = 152
- const Fl_Color **FL_DARK_CYAN** = 140
- const Fl_Color **FL_WHITE** = 255
- FL_EXPORT Fl_Color fl_inactive (Fl_Color c)

  Returns the inactive, dimmed version of the given color.
- FL_EXPORT Fl_Color fl_contrast (Fl_Color fg, Fl_Color bg)

  Returns a color that contrasts with the background color.
- FL_EXPORT Fl_Color fl_color_average (Fl_Color c1, Fl_Color c2, float weight)

  Returns the weighted average color between the two given colors.
- Fl_Color fl_lighter (Fl_Color c)

  Returns a lighter version of the specified color.
- Fl_Color fl_darker (Fl_Color c)

  Returns a darker version of the specified color.
- Fl_Color fl_rgb_color (uchar r, uchar g, uchar b)

  Returns the 24-bit color value closest to `r`, `g`, `b`.
- Fl_Color fl_rgb_color (uchar g)

  Returns the 24-bit color value closest to `g` (grayscale).
- Fl_Color fl_gray_ramp (int i)

  Returns a gray color value from black (i == 0) to white (i == FL_NUM_GRAY - 1).
- Fl_Color fl_color_cube (int r, int g, int b)

  Returns a color out of the color cube.

32.1.1 Detailed Description

This file contains type definitions and general enumerations.

32.1.2 Macro Definition Documentation

#define FL_ABI_VERSION (FL_MAJOR_VERSION*10000 + FL_MINOR_VERSION*100)

The FLTK ABI (Application Binary Interface) version number as an *int*.

FL_ABI_VERSION is an *int* that describes the major, minor, and patch ABI version numbers in the same format as FL_API_VERSION.

The ABI version number `FL_ABI_VERSION` is usually the same as the API version `FL_API_VERSION` with the last two digits set to '00'.

FLTK retains the ABI (Application Binary Interface) during patch releases of the same major and minor versions. Examples:

```
FLTK Version  FL_API_VERSION  FL_ABI_VERSION  FL_VERSION (deprecated)
  1.3.0          10300           10300           1.0300
  1.3.4          10304           10300           1.0304
```

Version 1.2.3 is actually stored as 10203 to allow for more than 9 minor and patch releases.

The FL_MAJOR_VERSION, FL_MINOR_VERSION, and FL_PATCH_VERSION constants give the integral values for the major, minor, and patch releases respectively.

To enable new ABI-breaking features in patch releases you can configure FLTK to use a higher FL_ABI_VERSION.

See Also

README.abi-version.txt

#define FL_API_VERSION (FL_MAJOR_VERSION*10000 + FL_MINOR_VERSION*100 + FL_PATCH_VERSION)

The FLTK API version number as an *int*.

FL_API_VERSION is an *int* that describes the major, minor, and patch version numbers.

Version 1.2.3 is actually stored as 10203 to allow for more than 9 minor and patch releases.

The FL_MAJOR_VERSION, FL_MINOR_VERSION, and FL_PATCH_VERSION constants give the integral values for the major, minor, and patch releases respectively.

Note

FL_API_VERSION is intended to replace the deprecated *double* FL_VERSION.

See Also

Fl::api_version()

#define FL_MAJOR_VERSION 1

The major release version of this FLTK library.

See Also

FL_VERSION

#define FL_MINOR_VERSION 3

The minor release version for this library.

FLTK remains mostly source-code compatible between minor version changes.

#define FL_PATCH_VERSION 4

The patch version for this library.

FLTK remains binary compatible between patches.

#define FL_VERSION

Value:

```
( (double)FL_MAJOR_VERSION + \
                          (double)FL_MINOR_VERSION * 0.01 + \
                          (double)FL_PATCH_VERSION * 0.0001 )
```

The FLTK version number as a *double*.

FL_VERSION is a *double* that describes the major, minor, and patch version numbers.

Version 1.2.3 is actually stored as 1.0203 to allow for more than 9 minor and patch releases.

Deprecated This `double` version number is retained for compatibility with existing program code. New code should use *int* FL_API_VERSION instead. FL_VERSION is deprecated because comparisons of floating point values may fail due to rounding errors. However, there are currently no plans to remove this deprecated constant.

FL_VERSION is equivalent to *(double)FL_API_VERSION / 10000*.

See Also

Fl::version() (deprecated as well)
FL_API_VERSION
Fl::api_version()

32.1.3 Typedef Documentation

typedef int Fl_Font

A font number is an index into the internal font table.

typedef int Fl_Fontsize

Size of a font in pixels.
This is the approximate height of a font in pixels.

32.1.4 Enumeration Type Documentation

anonymous enum

FD "when" conditions.

Enumerator

FL_READ Call the callback when there is data to be read.

FL_WRITE Call the callback when data can be written without blocking.

FL_EXCEPT Call the callback if an exception occurs on the file.

enum Fl_Boxtype

Enumerator

FL_NO_BOX nothing is drawn at all, this box is invisible

FL_FLAT_BOX a flat box

FL_UP_BOX see figure 1

FL_DOWN_BOX see figure 1

FL_UP_FRAME see figure 1

FL_DOWN_FRAME see figure 1

FL_THIN_UP_BOX see figure 1

FL_THIN_DOWN_BOX see figure 1

FL_THIN_UP_FRAME see figure 1

FL_THIN_DOWN_FRAME see figure 1

FL_ENGRAVED_BOX see figure 1

FL_EMBOSSED_BOX see figure 1

FL_ENGRAVED_FRAME see figure 1

FL_EMBOSSED_FRAME see figure 1

FL_BORDER_BOX see figure 1

_FL_SHADOW_BOX see figure 1

FL_BORDER_FRAME see figure 1

_FL_SHADOW_FRAME see figure 1

_FL_ROUNDED_BOX see figure 1

_FL_RSHADOW_BOX see figure 1

_FL_ROUNDED_FRAME see figure 1

_FL_RFLAT_BOX see figure 1

_FL_ROUND_UP_BOX see figure 1

_FL_ROUND_DOWN_BOX see figure 1

_FL_DIAMOND_UP_BOX see figure 1

_FL_DIAMOND_DOWN_BOX see figure 1

_FL_OVAL_BOX see figure 1

_FL_OSHADOW_BOX see figure 1

_FL_OVAL_FRAME see figure 1

_FL_OFLAT_BOX see figure 1

_FL_PLASTIC_UP_BOX plastic version of FL_UP_BOX

_FL_PLASTIC_DOWN_BOX plastic version of FL_DOWN_BOX

_FL_PLASTIC_UP_FRAME plastic version of FL_UP_FRAME

_FL_PLASTIC_DOWN_FRAME plastic version of FL_DOWN_FRAME

_FL_PLASTIC_THIN_UP_BOX plastic version of FL_THIN_UP_BOX

_FL_PLASTIC_THIN_DOWN_BOX plastic version of FL_THIN_DOWN_BOX

_FL_PLASTIC_ROUND_UP_BOX plastic version of FL_ROUND_UP_BOX

_FL_PLASTIC_ROUND_DOWN_BOX plastic version of FL_ROUND_DOWN_BOX

_FL_GTK_UP_BOX gtk+ version of FL_UP_BOX

_FL_GTK_DOWN_BOX gtk+ version of FL_DOWN_BOX

_FL_GTK_UP_FRAME gtk+ version of FL_UP_FRAME

_FL_GTK_DOWN_FRAME gtk+ version of FL_DOWN_FRAME

_FL_GTK_THIN_UP_BOX gtk+ version of FL_THIN_UP_BOX

_FL_GTK_THIN_DOWN_BOX gtk+ version of FL_THIN_DOWN_BOX

_FL_GTK_THIN_UP_FRAME gtk+ version of FL_THIN_UP_FRAME

_FL_GTK_THIN_DOWN_FRAME gtk+ version of FL_THIN_DOWN_FRAME

_FL_GTK_ROUND_UP_BOX gtk+ version of FL_ROUND_UP_BOX

_FL_GTK_ROUND_DOWN_BOX gtk+ version of FL_ROUND_DOWN_BOX

_FL_GLEAM_UP_BOX gleam version of FL_UP_BOX

_FL_GLEAM_DOWN_BOX gleam version of FL_DOWN_BOX

_FL_GLEAM_UP_FRAME gleam version of FL_UP_FRAME

_FL_GLEAM_DOWN_FRAME gleam version of FL_DOWN_FRAME

_FL_GLEAM_THIN_UP_BOX gleam version of FL_THIN_UP_BOX

_FL_GLEAM_THIN_DOWN_BOX gleam version of FL_THIN_DOWN_BOX

_FL_GLEAM_ROUND_UP_BOX gleam version of FL_ROUND_UP_BOX

_FL_GLEAM_ROUND_DOWN_BOX gleam version of FL_ROUND_DOWN_BOX

FL_FREE_BOXTYPE the first free box type for creation of new box types

enum Fl_Cursor

The following constants define the mouse cursors that are available in FLTK.
Cursors are provided by the system when available, or bitmaps built into FLTK as a fallback.

Todo enum Fl_Cursor needs maybe an image.

Enumerator

FL_CURSOR_DEFAULT the default cursor, usually an arrow.

FL_CURSOR_ARROW an arrow pointer.

FL_CURSOR_CROSS crosshair.

FL_CURSOR_WAIT busy indicator (e.g. hourglass).

FL_CURSOR_INSERT I-beam.

FL_CURSOR_HAND pointing hand.

FL_CURSOR_HELP question mark pointer.

FL_CURSOR_MOVE 4-pointed arrow or hand.

FL_CURSOR_NS up/down resize.

FL_CURSOR_WE left/right resize.

FL_CURSOR_NWSE diagonal resize.

FL_CURSOR_NESW diagonal resize.

FL_CURSOR_N upwards resize.

FL_CURSOR_NE upwards, right resize.

FL_CURSOR_E rightwards resize.

FL_CURSOR_SE downwards, right resize.

FL_CURSOR_S downwards resize.

FL_CURSOR_SW downwards, left resize.

FL_CURSOR_W leftwards resize.

FL_CURSOR_NW upwards, left resize.

FL_CURSOR_NONE invisible.

enum Fl_Damage

Damage masks.

Enumerator

FL_DAMAGE_CHILD A child needs to be redrawn.

FL_DAMAGE_EXPOSE The window was exposed.

FL_DAMAGE_SCROLL The Fl_Scroll widget was scrolled.

FL_DAMAGE_OVERLAY The overlay planes need to be redrawn.

FL_DAMAGE_USER1 First user-defined damage bit.

FL_DAMAGE_USER2 Second user-defined damage bit.

FL_DAMAGE_ALL Everything needs to be redrawn.

enum Fl_Event

Every time a user moves the mouse pointer, clicks a button, or presses a key, an event is generated and sent to your application.

Events can also come from other programs like the window manager.

Events are identified by the integer argument passed to the Fl_Widget::handle() virtual method. Other information about the most recent event is stored in static locations and acquired by calling the Fl::event_*() methods. This static information remains valid until the next event is read from the window system, so it is ok to look at it outside of the handle() method.

Event numbers can be converted to their actual names using the fl_eventnames[] array defined in #include <FL/names.h>

See Also

Fl::event_text(), Fl::event_key(), class Fl::

Enumerator

FL_NO_EVENT No event.

FL_PUSH A mouse button has gone down with the mouse pointing at this widget. You can find out what button by calling Fl::event_button(). You find out the mouse position by calling Fl::event_x() and Fl::event_y().

A widget indicates that it "wants" the mouse click by returning non-zero from its Fl_Widget::handle() method. It will then become the Fl::pushed() widget and will get FL_DRAG and the matching FL_RELEASE events. If Fl_Widget::handle() returns zero then FLTK will try sending the FL_PUSH to another widget.

FL_RELEASE A mouse button has been released. You can find out what button by calling Fl::event_button().

In order to receive the FL_RELEASE event, the widget must return non-zero when handling FL_PUSH.

FL_ENTER The mouse has been moved to point at this widget. This can be used for highlighting feedback. If a widget wants to highlight or otherwise track the mouse, it indicates this by returning non-zero from its handle() method. It then becomes the Fl::belowmouse() widget and will receive FL_MOVE and FL_LEAVE events.

FL_LEAVE The mouse has moved out of the widget. In order to receive the FL_LEAVE event, the widget must return non-zero when handling FL_ENTER.

FL_DRAG The mouse has moved with a button held down. The current button state is in Fl::event_state(). The mouse position is in Fl::event_x() and Fl::event_y().

In order to receive FL_DRAG events, the widget must return non-zero when handling FL_PUSH.

FL_FOCUS This indicates an *attempt* to give a widget the keyboard focus. If a widget wants the focus, it should change itself to display the fact that it has the focus, and return non-zero from its handle() method. It then becomes the Fl::focus() widget and gets FL_KEYDOWN, FL_KEYUP, and FL_UNFOCUS events.

The focus will change either because the window manager changed which window gets the focus, or because the user tried to navigate using tab, arrows, or other keys. You can check Fl::event_key() to figure out why it moved. For navigation it will be the key pressed and for interaction with the window manager it will be zero.

FL_UNFOCUS This event is sent to the previous Fl::focus() widget when another widget gets the focus or the window loses focus.

FL_KEYDOWN A key was pressed (FL_KEYDOWN) or released (FL_KEYUP). Fl_KEYBOARD is a synonym for FL_KEYDOWN. The key can be found in Fl::event_key(). The text that the key should insert can be found with Fl::event_text() and its length is in Fl::event_length(). If you use the key handle() should return 1. If you return zero then FLTK assumes you ignored the key and will then attempt to send it to a parent widget. If none of them want it, it will change the event into a FL_SHORTCUT event.

To receive FL_KEYBOARD events you must also respond to the FL_FOCUS and FL_UNFOCUS events.

If you are writing a text-editing widget you may also want to call the Fl::compose() function to translate individual keystrokes into non-ASCII characters.

FL_KEYUP events are sent to the widget that currently has focus. This is not necessarily the same widget that received the corresponding FL_KEYDOWN event because focus may have changed between events.

FL_KEYBOARD Equivalent to FL_KEYDOWN.

See Also

FL_KEYDOWN

FL_KEYUP Key release event.

See Also

FL_KEYDOWN

FL_CLOSE The user clicked the close button of a window. This event is used internally only to trigger the callback of Fl_Window derived classed. The default callback closes the window calling Fl_Window::hide().

FL_MOVE The mouse has moved without any mouse buttons held down. This event is sent to the Fl::belowmouse() widget.

In order to receive FL_MOVE events, the widget must return non-zero when handling FL_ENTER.

FL_SHORTCUT If the Fl::focus() widget is zero or ignores an FL_KEYBOARD event then FLTK tries sending this event to every widget it can, until one of them returns non-zero. FL_SHORTCUT is first sent to the Fl::belowmouse() widget, then its parents and siblings, and eventually to every widget in the window, trying to find an object that returns non-zero. FLTK tries really hard to not to ignore any keystrokes!

You can also make ”global” shortcuts by using Fl::add_handler(). A global shortcut will work no matter what windows are displayed or which one has the focus.

FL_DEACTIVATE This widget is no longer active, due to Fl_Widget::deactivate() being called on it or one of its parents. Fl_Widget::active() may still be true after this, the widget is only active if Fl_Widget::active() is true on it and all its parents (use Fl_Widget::active_r() to check this).

FL_ACTIVATE This widget is now active, due to Fl_Widget::activate() being called on it or one of its parents.

FL_HIDE This widget is no longer visible, due to Fl_Widget::hide() being called on it or one of its parents, or due to a parent window being minimized. Fl_Widget::visible() may still be true after this, but the widget is visible only if visible() is true for it and all its parents (use Fl_Widget::visible_r() to check this).

FL_SHOW This widget is visible again, due to Fl_Widget::show() being called on it or one of its parents, or due to a parent window being restored. Child Fl_Windows respond to this by actually creating the window if not done already, so if you subclass a window, be sure to pass FL_SHOW to the base class Fl_Widget::handle() method!

FL_PASTE You should get this event some time after you call Fl::paste(). The contents of Fl::event_text() is the text to insert and the number of characters is in Fl::event_length().

FL_SELECTIONCLEAR The Fl::selection_owner() will get this event before the selection is moved to another widget. This indicates that some other widget or program has claimed the selection. Motif programs used this to clear the selection indication. Most modern programs ignore this.

FL_MOUSEWHEEL The user has moved the mouse wheel. The Fl::event_dx() and Fl::event_dy() methods can be used to find the amount to scroll horizontally and vertically.

FL_DND_ENTER The mouse has been moved to point at this widget. A widget that is interested in receiving drag'n'drop data must return 1 to receive FL_DND_DRAG, FL_DND_LEAVE and FL_DND_RELEASE events.

FL_DND_DRAG The mouse has been moved inside a widget while dragging data. A widget that is interested in receiving drag'n'drop data should indicate the possible drop position.

FL_DND_LEAVE The mouse has moved out of the widget.

FL_DND_RELEASE The user has released the mouse button dropping data into the widget. If the widget returns 1, it will receive the data in the immediately following FL_PASTE event.

FL_SCREEN_CONFIGURATION_CHANGED The screen configuration (number, positions) was changed. Use Fl::add_handler() to be notified of this event.

FL_FULLSCREEN The fullscreen state of the window has changed.

FL_ZOOM_GESTURE The user has made a zoom/pinch/magnification gesture. The Fl::event_dy() method can be used to find magnification amount, Fl::event_x() and Fl::event_y() are set as well.

enum Fl_Labeltype

The labeltype() method sets the type of the label.

The following standard label types are included:

Todo The doxygen comments are incomplete, and some labeltypes start with an underscore. Also, there are three external functions undocumented (yet):

- fl_define_FL_SHADOW_LABEL()
- fl_define_FL_ENGRAVED_LABEL()
- fl_define_FL_EMBOSSED_LABEL()

Enumerator

FL_NORMAL_LABEL draws the text (0)

FL_NO_LABEL does nothing

_FL_SHADOW_LABEL draws a drop shadow under the text

_FL_ENGRAVED_LABEL draws edges as though the text is engraved

_FL_EMBOSSED_LABEL draws edges as though the text is raised

_FL_MULTI_LABEL draws a composite label

See Also

Fl_Multi_Label

_FL_ICON_LABEL draws the icon associated with the text

_FL_IMAGE_LABEL the label displays an "icon" based on a Fl_Image

FL_FREE_LABELTYPE first free labeltype to use for creating own labeltypes

enum Fl_When

These constants determine when a callback is performed.

See Also

Fl_Widget::when();

Todo doxygen comments for values are incomplete and maybe wrong or unclear

Enumerator

FL_WHEN_NEVER Never call the callback.

FL_WHEN_CHANGED Do the callback only when the widget value changes.

FL_WHEN_NOT_CHANGED Do the callback whenever the user interacts with the widget.

FL_WHEN_RELEASE Do the callback when the button or key is released and the value changes.

FL_WHEN_RELEASE_ALWAYS Do the callback when the button or key is released, even if the value doesn't change.

FL_WHEN_ENTER_KEY Do the callback when the user presses the ENTER key and the value changes.

FL_WHEN_ENTER_KEY_ALWAYS Do the callback when the user presses the ENTER key, even if the value doesn't change.

FL_WHEN_ENTER_KEY_CHANGED ?

32.1.5 Function Documentation

Fl_Boxtype fl_box (Fl_Boxtype *b*) `[inline]`

Get the filled version of a frame.

If no filled version of a given frame exists, the behavior of this function is undefined and some random box or frame is returned.

Fl_Color fl_color_cube (int *r*, int *g*, int *b*) `[inline]`

Returns a color out of the color cube.

`r` must be in the range 0 to FL_NUM_RED (5) minus 1, `g` must be in the range 0 to FL_NUM_GREEN (8) minus 1, `b` must be in the range 0 to FL_NUM_BLUE (5) minus 1.

To get the closest color to a 8-bit set of R,G,B values use:

```
fl_color_cube(R * (FL_NUM_RED - 1) / 255,
   G * (FL_NUM_GREEN - 1) / 255,
   B * (FL_NUM_BLUE - 1) / 255);
```

Fl_Color fl_darker (Fl_Color *c*) `[inline]`

Returns a darker version of the specified color.

Fl_Boxtype fl_down (Fl_Boxtype *b*) `[inline]`

Get the "pressed" or "down" version of a box.

If no "down" version of a given box exists, the behavior of this function is undefined and some random box or frame is returned.

Fl_Boxtype fl_frame (Fl_Boxtype *b*) `[inline]`

Get the unfilled, frame only version of a box.

If no frame version of a given box exists, the behavior of this function is undefined and some random box or frame is returned.

Fl_Color fl_gray_ramp (int *i*) `[inline]`

Returns a gray color value from black (i == 0) to white (i == FL_NUM_GRAY - 1).

FL_NUM_GRAY is defined to be 24 in the current FLTK release. To get the closest FLTK gray value to an 8-bit grayscale color 'I' use:

```
fl_gray_ramp(I * (FL_NUM_GRAY - 1) / 255)
```

Fl_Color fl_lighter (Fl_Color *c*) `[inline]`

Returns a lighter version of the specified color.

Fl_Color fl_rgb_color (uchar *r*, uchar *g*, uchar *b*) `[inline]`

Returns the 24-bit color value closest to `r`, `g`, `b`.

Fl_Color fl_rgb_color (uchar *g*) `[inline]`

Returns the 24-bit color value closest to `g` (grayscale).

32.1.6 Variable Documentation

const Fl_Align FL_ALIGN_BOTTOM = (Fl_Align)2

Align the label at the bottom of the widget.

const Fl_Align FL_ALIGN_CENTER = (Fl_Align)0

Align the label horizontally in the middle.

const Fl_Align FL_ALIGN_CLIP = (Fl_Align)64

All parts of the label that are lager than the widget will not be drawn .

const Fl_Align FL_ALIGN_IMAGE_BACKDROP = (Fl_Align)0x0200

If the label contains an image, draw the image or deimage in the background.

const Fl_Align FL_ALIGN_IMAGE_NEXT_TO_TEXT = (Fl_Align)0x0100

If the label contains an image, draw the text to the right of the image.

const Fl_Align FL_ALIGN_IMAGE_OVER_TEXT = (Fl_Align)0x0000

If the label contains an image, draw the text below the image.

const Fl_Align FL_ALIGN_INSIDE = (Fl_Align)16

Draw the label inside of the widget.

const Fl_Align FL_ALIGN_LEFT = (Fl_Align)4

Align the label at the left of the widget.

Inside labels appear left-justified starting at the left side of the widget, outside labels are right-justified and drawn to the left of the widget.

const Fl_Align FL_ALIGN_RIGHT = (Fl_Align)8

Align the label to the right of the widget.

const Fl_Align FL_ALIGN_TEXT_NEXT_TO_IMAGE = (Fl_Align)0x0120

If the label contains an image, draw the text to the left of the image.

const Fl_Align FL_ALIGN_TEXT_OVER_IMAGE = (Fl_Align)0x0020

If the label contains an image, draw the text on top of the image.

const Fl_Align FL_ALIGN_TOP = (Fl_Align)1

Align the label at the top of the widget.

Inside labels appear below the top, outside labels are drawn on top of the widget.

const Fl_Align FL_ALIGN_WRAP = (Fl_Align)128

Wrap text that does not fit the width of the widget.

FL_EXPORT Fl_Fontsize FL_NORMAL_SIZE

normal font size

normal font size

32.2 filename.H File Reference

File names and URI utility functions.

```
   #include "Fl_Export.H"
#include <sys/types.h>
#include <dirent.h>
```

Macros

- #define **fl_dirent_h_cyclic_include**
- #define **FL_FILENAME_H**
- #define FL_PATH_MAX 2048

  all path buffers should use this length

Typedefs

- typedef int(Fl_File_Sort_F)(struct dirent ∗∗, struct dirent ∗∗)

  File sorting function.

Functions

- FL_EXPORT void fl_decode_uri (char ∗uri)

  Decodes a URL-encoded string.
- FL_EXPORT int fl_filename_absolute (char ∗to, int tolen, const char ∗from)

  Makes a filename absolute from a relative filename.
- FL_EXPORT int fl_filename_expand (char ∗to, int tolen, const char ∗from)

  Expands a filename containing shell variables and tilde (∼).
- FL_EXPORT const char ∗ fl_filename_ext (const char ∗buf)

  Gets the extensions of a filename.
- FL_EXPORT void fl_filename_free_list (struct dirent ∗∗∗l, int n)

  Free the list of filenames that is generated by fl_filename_list().
- FL_EXPORT int fl_filename_isdir (const char ∗name)

  Determines if a file exists and is a directory from its filename.
- FL_EXPORT int fl_filename_list (const char ∗d, struct dirent ∗∗∗l, Fl_File_Sort_F ∗s=fl_numericsort)

  Portable and const-correct wrapper for the scandir() function.
- FL_EXPORT int fl_filename_match (const char ∗name, const char ∗pattern)

  Checks if a string `s` matches a pattern `p`.
- FL_EXPORT const char ∗ fl_filename_name (const char ∗filename)

  Gets the file name from a path.
- FL_EXPORT int fl_filename_relative (char ∗to, int tolen, const char ∗from)

  Makes a filename relative to the current working directory.
- FL_EXPORT char ∗ fl_filename_setext (char ∗to, int tolen, const char ∗ext)

  Replaces the extension in `buf` of max.
- FL_EXPORT int fl_open_uri (const char ∗uri, char ∗msg, int msglen)

  Opens the specified Uniform Resource Identifier (URI).

32.2.1 Detailed Description

File names and URI utility functions.

32.3 Fl.H File Reference

Fl static class.

```
  #include <FL/Fl_Export.H>
#include <FL/Fl_Cairo.H>
#include "fl_utf8.h"
#include "Enumerations.H"
```

Classes

- class Fl

  The Fl is the FLTK global (static) class containing state information and global methods for the current application.
- class Fl_Widget_Tracker

  This class should be used to control safe widget deletion.

Macros

- #define Fl_Object Fl_Widget

  for back compatibility - use Fl_Widget!
- #define **FL_SOCKET** int

Typedefs

- typedef void(∗ Fl_Abort_Handler)(const char ∗format,...)

  Signature of set_abort functions passed as parameters.
- typedef int(∗ Fl_Args_Handler)(int argc, char ∗∗argv, int &i)

  Signature of args functions passed as parameters.
- typedef void(∗ Fl_Atclose_Handler)(Fl_Window ∗window, void ∗data)

  Signature of set_atclose functions passed as parameters.
- typedef void(∗ Fl_Awake_Handler)(void ∗data)

  Signature of some wakeup callback functions passed as parameters.
- typedef void(Fl_Box_Draw_F)(int x, int y, int w, int h, Fl_Color color)

  Signature of some box drawing functions passed as parameters.
- typedef void(∗ Fl_Clipboard_Notify_Handler)(int source, void ∗data)

  Signature of add_clipboard_notify functions passed as parameters.
- typedef int(∗ Fl_Event_Dispatch)(int event, Fl_Window ∗w)

  Signature of event_dispatch functions passed as parameters.
- typedef int(∗ Fl_Event_Handler)(int event)

  Signature of add_handler functions passed as parameters.
- typedef void(∗ Fl_FD_Handler)(FL_SOCKET fd, void ∗data)

  Signature of add_fd functions passed as parameters.
- typedef void(∗ Fl_Idle_Handler)(void ∗data)

  Signature of add_idle callback functions passed as parameters.
- typedef void(Fl_Label_Draw_F)(const Fl_Label ∗label, int x, int y, int w, int h, Fl_Align align)

  Signature of some label drawing functions passed as parameters.
- typedef void(Fl_Label_Measure_F)(const Fl_Label ∗label, int &width, int &height)

  Signature of some label measurement functions passed as parameters.
- typedef void(∗ Fl_Old_Idle_Handler)()

  Signature of set_idle callback functions passed as parameters.
- typedef int(∗ Fl_System_Handler)(void ∗event, void ∗data)

  Signature of add_system_handler functions passed as parameters.
- typedef void(∗ Fl_Timeout_Handler)(void ∗data)

  Signature of some timeout callback functions passed as parameters.

Variables

- FL_EXPORT const char ∗ fl_local_alt

  string pointer used in shortcuts, you can change it to another language
- FL_EXPORT const char ∗ fl_local_ctrl

  string pointer used in shortcuts, you can change it to another language
- FL_EXPORT const char ∗ fl_local_meta

  string pointer used in shortcuts, you can change it to another language
- FL_EXPORT const char ∗ fl_local_shift

  string pointer used in shortcuts, you can change it to another language

32.3.1 Detailed Description

Fl static class.

32.4 fl_arc.cxx File Reference

Utility functions for drawing arcs and circles.

```
   #include <FL/fl_draw.H>
#include <FL/math.h>
```

32.4.1 Detailed Description

Utility functions for drawing arcs and circles.

32.5 fl_arci.cxx File Reference

Utility functions for drawing circles using integers.

```
   #include <FL/fl_draw.H>
#include <FL/x.H>
#include <config.h>
```

32.5.1 Detailed Description

Utility functions for drawing circles using integers.

32.6 fl_ask.cxx File Reference

Utility functions for common dialogs.

```
   #include <stdio.h>
#include <stdarg.h>
#include "flstring.h"
#include <FL/Fl.H>
#include <FL/fl_ask.H>
#include <FL/Fl_Box.H>
#include <FL/Fl_Button.H>
#include <FL/Fl_Return_Button.H>
#include <FL/Fl_Window.H>
#include <FL/Fl_Input.H>
#include <FL/Fl_Secret_Input.H>
#include <FL/x.H>
#include <FL/fl_draw.H>
```

Functions

- void fl_alert (const char ∗fmt,...)

  Shows an alert message dialog box.
- int fl_ask (const char ∗fmt,...)

  Shows a dialog displaying the `fmt` *message, this dialog features 2 yes/no buttons.*
- void fl_beep (int type)

Emits a system beep message.

- int fl_choice (const char ∗fmt, const char ∗b0, const char ∗b1, const char ∗b2,...)

  Shows a dialog displaying the printf style `fmt` *message, this dialog features up to 3 customizable choice buttons.*
- const char ∗ fl_input (const char ∗fmt, const char ∗defstr,...)

  Shows an input dialog displaying the `fmt` *message.*
- void fl_message (const char ∗fmt,...)

  Shows an information message dialog box.
- void fl_message_hotspot (int enable)

  Sets whether or not to move the common message box used in many common dialogs like fl_message(), fl_alert(), fl_ask(), fl_choice(), fl_input(), fl_password() to follow the mouse pointer.
- int fl_message_hotspot (void)

  Gets whether or not to move the common message box used in many common dialogs like fl_message(), fl_alert(), fl_ask(), fl_choice(), fl_input(), fl_password() to follow the mouse pointer.
- Fl_Widget ∗ fl_message_icon ()

  Gets the Fl_Box icon container of the current default dialog used in many common dialogs like fl_message(), fl_alert(), fl_ask(), fl_choice(), fl_input(), fl_password()
- void fl_message_title (const char ∗title)

  Sets the title of the dialog window used in many common dialogs.
- void fl_message_title_default (const char ∗title)

  Sets the default title of the dialog window used in many common dialogs.
- const char ∗ fl_password (const char ∗fmt, const char ∗defstr,...)

  Shows an input dialog displaying the `fmt` *message.*

Variables

- const char ∗ fl_cancel = ”Cancel”

  string pointer used in common dialogs, you can change it to another language
- const char ∗ fl_close = ”Close”

  string pointer used in common dialogs, you can change it to another language
- Fl_Font **fl_message_font_** = FL_HELVETICA
- Fl_Fontsize **fl_message_size_** = -1
- const char ∗ fl_no = ”No”

  string pointer used in common dialogs, you can change it to another language
- const char ∗ fl_ok = ”OK”

  string pointer used in common dialogs, you can change it to another language
- const char ∗ fl_yes = ”Yes”

  string pointer used in common dialogs, you can change it to another language

32.6.1 Detailed Description

Utility functions for common dialogs.

32.7 fl_ask.H File Reference

API for common dialogs.

```
#include "Enumerations.H"
```

Macros

- #define **__fl_attr**(x)

Enumerations

- enum Fl_Beep {
FL_BEEP_DEFAULT = 0, FL_BEEP_MESSAGE, FL_BEEP_ERROR, FL_BEEP_QUESTION,
FL_BEEP_PASSWORD, FL_BEEP_NOTIFICATION }

  Different system beeps available.

Functions

- FL_EXPORT void FL_EXPORT void **fl_alert** (const char ∗,...) __fl_attr((__format__(__printf__
- FL_EXPORT void FL_EXPORT void
FL_EXPORT int **fl_ask** (const char ∗,...) __fl_attr((__format__(__printf__
- FL_EXPORT void fl_beep (int type=FL_BEEP_DEFAULT)

  Emits a system beep message.
- FL_EXPORT int **fl_choice** (const char ∗q, const char ∗b0, const char ∗b1, const char ∗b2,...) __fl_attr((__format__(__printf__
- FL_EXPORT int FL_EXPORT const
char ∗ **fl_input** (const char ∗label, const char ∗deflt=0,...) __fl_attr((__format__(__printf__
- FL_EXPORT void **fl_message** (const char ∗,...) __fl_attr((__format__(__printf__
- void **fl_message_font** (Fl_Font f, Fl_Fontsize s)
- FL_EXPORT void fl_message_hotspot (int enable)

  Sets whether or not to move the common message box used in many common dialogs like fl_message(), fl_alert(), fl_ask(), fl_choice(), fl_input(), fl_password() to follow the mouse pointer.
- FL_EXPORT int fl_message_hotspot (void)

  Gets whether or not to move the common message box used in many common dialogs like fl_message(), fl_alert(), fl_ask(), fl_choice(), fl_input(), fl_password() to follow the mouse pointer.
- FL_EXPORT int FL_EXPORT const
char FL_EXPORT const char
FL_EXPORT Fl_Widget ∗ fl_message_icon ()

  Gets the Fl_Box icon container of the current default dialog used in many common dialogs like fl_message(), fl_alert(), fl_ask(), fl_choice(), fl_input(), fl_password()
- FL_EXPORT void fl_message_title (const char ∗title)

  Sets the title of the dialog window used in many common dialogs.
- FL_EXPORT void fl_message_title_default (const char ∗title)

  Sets the default title of the dialog window used in many common dialogs.
- FL_EXPORT int FL_EXPORT const
char FL_EXPORT const char ∗ **fl_password** (const char ∗label, const char ∗deflt=0,...) __fl_attr((__format__(__printf__

Variables

- FL_EXPORT void FL_EXPORT void
FL_EXPORT int **__deprecated__**
- FL_EXPORT const char ∗ fl_cancel

  string pointer used in common dialogs, you can change it to another language
- FL_EXPORT const char ∗ fl_close

  string pointer used in common dialogs, you can change it to another language

- FL_EXPORT Fl_Font **fl_message_font_**
- FL_EXPORT Fl_Fontsize **fl_message_size_**
- FL_EXPORT const char ∗ fl_no

  string pointer used in common dialogs, you can change it to another language
- FL_EXPORT const char ∗ fl_ok

  string pointer used in common dialogs, you can change it to another language
- FL_EXPORT const char ∗ fl_yes

  string pointer used in common dialogs, you can change it to another language

32.7.1 Detailed Description

API for common dialogs.

32.7.2 Enumeration Type Documentation

enum Fl_Beep

Different system beeps available.

See Also

fl_beep(int)

Enumerator

FL_BEEP_DEFAULT Default beep.

FL_BEEP_MESSAGE Message beep.

FL_BEEP_ERROR Error beep.

FL_BEEP_QUESTION Question beep.

FL_BEEP_PASSWORD Password beep.

FL_BEEP_NOTIFICATION Notification beep.

32.8 fl_boxtype.cxx File Reference

drawing code for common box types.

```
  #include <FL/Fl.H>
#include <FL/Fl_Widget.H>
#include <FL/fl_draw.H>
#include <config.h>
```

Macros

- #define **D1** BORDER_WIDTH
- #define **D2** (BORDER_WIDTH+BORDER_WIDTH)
- #define fl_border_box fl_rectbound

  allow consistent naming

Functions

- void fl_border_frame (int x, int y, int w, int h, Fl_Color c)

  Draws a frame of type FL_BORDER_FRAME.
- void fl_down_box (int x, int y, int w, int h, Fl_Color c)

  Draws a box of type FL_DOWN_BOX.
- void fl_down_frame (int x, int y, int w, int h, Fl_Color)

  Draws a frame of type FL_DOWN_FRAME.
- void fl_draw_box (Fl_Boxtype t, int x, int y, int w, int h, Fl_Color c)

  Draws a box using given type, position, size and color.
- void fl_embossed_box (int x, int y, int w, int h, Fl_Color c)

  Draws a box of type FL_EMBOSSED_BOX.
- void fl_embossed_frame (int x, int y, int w, int h, Fl_Color)

  Draws a frame of type FL_EMBOSSED_FRAME.
- void fl_engraved_box (int x, int y, int w, int h, Fl_Color c)

  Draws a box of type FL_ENGRAVED_BOX.
- void fl_engraved_frame (int x, int y, int w, int h, Fl_Color)

  Draws a frame of type FL_ENGRAVED_FRAME.
- void fl_flat_box (int x, int y, int w, int h, Fl_Color c)

  Draws a box of type FL_FLAT_BOX.
- void fl_frame (const char ∗s, int x, int y, int w, int h)

  Draws a series of line segments around the given box.
- void fl_frame2 (const char ∗s, int x, int y, int w, int h)

  Draws a series of line segments around the given box.
- const uchar ∗ **fl_gray_ramp** ()
- void fl_internal_boxtype (Fl_Boxtype t, Fl_Box_Draw_F ∗f)

  Sets the drawing function for a given box type.
- void fl_no_box (int, int, int, int, Fl_Color)

  Draws a box of type FL_NO_BOX.
- void fl_rectbound (int x, int y, int w, int h, Fl_Color bgcolor)

  Draws a bounded rectangle with a given position, size and color.
- void fl_thin_down_box (int x, int y, int w, int h, Fl_Color c)

  Draws a box of type FL_THIN_DOWN_BOX.
- void fl_thin_down_frame (int x, int y, int w, int h, Fl_Color)

  Draws a frame of type FL_THIN_DOWN_FRAME.
- void fl_thin_up_box (int x, int y, int w, int h, Fl_Color c)

  Draws a box of type FL_THIN_UP_BOX.
- void fl_thin_up_frame (int x, int y, int w, int h, Fl_Color)

  Draws a frame of type FL_THIN_UP_FRAME.
- void fl_up_box (int x, int y, int w, int h, Fl_Color c)

  Draws a box of type FL_UP_BOX.
- void fl_up_frame (int x, int y, int w, int h, Fl_Color)

  Draws a frame of type FL_UP_FRAME.

32.8.1 Detailed Description

drawing code for common box types.

32.8.2 Function Documentation

void fl_internal_boxtype (Fl_Boxtype *t*, Fl_Box_Draw_F ∗ *f*)

Sets the drawing function for a given box type.

Parameters

`in`	*t*	box type
`in`	*f*	box drawing function

void fl_rectbound (int *x*, int *y*, int *w*, int *h*, Fl_Color *bgcolor*)

Draws a bounded rectangle with a given position, size and color.
Equivalent to drawing a box of type FL_BORDER_BOX.

32.9 fl_color.cxx File Reference

Color handling.

```
#include "Fl_XColor.H"
#include <FL/Fl.H>
#include <FL/x.H>
#include <FL/fl_draw.H>
#include "fl_cmap.h"
```

Macros

- #define fl_overlay 0

  HAVE_OVERLAY determines whether fl_overlay is variable or defined as 0.

Functions

- Fl_Color fl_color_average (Fl_Color color1, Fl_Color color2, float weight)

  Returns the weighted average color between the two given colors.
- Fl_Color fl_contrast (Fl_Color fg, Fl_Color bg)

  Returns a color that contrasts with the background color.
- Fl_Color fl_inactive (Fl_Color c)

  Returns the inactive, dimmed version of the given color.
- ulong fl_xpixel (uchar r, uchar g, uchar b)

  Returns the X pixel number used to draw the given rgb color.
- ulong fl_xpixel (Fl_Color i)

  Returns the X pixel number used to draw the given FLTK color index.

Variables

- uchar fl_bluemask

  color mask used in current color map handling
- int fl_blueshift

  color shift used in current color map handling
- int fl_extrashift

  color shift used in current color map handling
- uchar fl_greenmask

  color mask used in current color map handling
- int fl_greenshift

  color shift used in current color map handling

- uchar fl_redmask

  color mask used in current color map handling
- int fl_redshift

  color shift used in current color map handling
- Fl_XColor fl_xmap [1][256]

  HAVE_OVERLAY determines whether fl_xmap is one or two planes.

32.9.1 Detailed Description

Color handling.

32.10 Fl_Color_Chooser.H File Reference

Fl_Color_Chooser widget .

```
   #include <FL/Fl_Group.H>
#include <FL/Fl_Box.H>
#include <FL/Fl_Return_Button.H>
#include <FL/Fl_Choice.H>
#include <FL/Fl_Value_Input.H>
```

Classes

- class Fl_Color_Chooser

  The Fl_Color_Chooser widget provides a standard RGB color chooser.

32.10.1 Detailed Description

Fl_Color_Chooser widget .

32.11 Fl_compose.cxx File Reference

Utility functions to support text input.

```
   #include <FL/Fl.H>
#include <FL/x.H>
```

Variables

- XIC **fl_xim_ic**

32.11.1 Detailed Description

Utility functions to support text input.

32.12 fl_curve.cxx File Reference

Utility for drawing Bezier curves, adding the points to the current fl_begin/fl_vertex/fl_end path.

```
   #include <FL/fl_draw.H>
#include <math.h>
```

32.12.1 Detailed Description

Utility for drawing Bezier curves, adding the points to the current fl_begin/fl_vertex/fl_end path. Incremental math implementation: I very much doubt this is optimal! From Foley/vanDam page 511. If anybody has a better algorithm, please send it!

32.13 Fl_Device.H File Reference

declaration of classes Fl_Device, Fl_Graphics_Driver, Fl_Surface_Device, Fl_Display_Device, Fl_Device_-Plugin.

```
  #include <FL/x.H>
#include <FL/Fl_Plugin.H>
#include <FL/Fl_Image.H>
#include <FL/Fl_Bitmap.H>
#include <FL/Fl_Pixmap.H>
#include <FL/Fl_RGB_Image.H>
#include <stdlib.h>
```

Classes

- class Fl_Device

    All graphical output devices and all graphics systems.
- class Fl_Device_Plugin

    This plugin socket allows the integration of new device drivers for special window or screen types.
- class Fl_Display_Device

    A display to which the computer can draw.
- class Fl_GDI_Graphics_Driver

    The MSWindows-specific graphics class.
- class Fl_GDI_Printer_Graphics_Driver

    The graphics driver used when printing on MSWindows.
- class Fl_Graphics_Driver

    A virtual class subclassed for each graphics driver FLTK uses.
- class Fl_Quartz_Graphics_Driver

    The Mac OS X-specific graphics class.
- class Fl_Surface_Device

    A drawing surface that's susceptible to receive graphical output.
- class Fl_Xlib_Graphics_Driver

    The Xlib-specific graphics class.
- struct Fl_Graphics_Driver::matrix

    A 2D coordinate transformation matrix.

Macros

- #define **FL_MATRIX_STACK_SIZE** 32
- #define **FL_REGION_STACK_SIZE** 10
- #define **XPOINT** XPoint

Typedefs

- typedef short **COORD_T**
- typedef void(∗ Fl_Draw_Image_Cb)(void ∗data, int x, int y, int w, uchar ∗buf)

    signature of image generation callback function.

Variables

- FL_EXPORT Fl_Graphics_Driver ∗ fl_graphics_driver

    Points to the driver that currently receives all graphics requests.

32.13.1 Detailed Description

declaration of classes Fl_Device, Fl_Graphics_Driver, Fl_Surface_Device, Fl_Display_Device, Fl_Device_-Plugin.

32.13.2 Typedef Documentation

typedef void(∗ Fl_Draw_Image_Cb)(void ∗data, int x, int y, int w, uchar ∗buf)

signature of image generation callback function.
Parameters

in	*data*	user data passed to function
in	*x,y,w*	position and width of scan line in image
out	*buf*	buffer for generated image data. You must copy `w` pixels from scanline `y`, starting at pixel `x` to this buffer.

32.14 Fl_Double_Window.cxx File Reference

Fl_Double_Window implementation.

```
   #include <config.h>
#include <FL/Fl.H>
#include <FL/Fl_Double_Window.H>
#include <FL/Fl_Overlay_Window.H>
#include <FL/Fl_Printer.H>
#include <FL/x.H>
#include <FL/fl_draw.H>
```

Functions

- void fl_begin_offscreen (Fl_Offscreen ctx)

    Send all subsequent drawing commands to this offscreen buffer.
- char fl_can_do_alpha_blending ()

    Checks whether platform supports true alpha blending for RGBA images.
- void fl_copy_offscreen (int x, int y, int w, int h, Fl_Offscreen pixmap, int srcx, int srcy)

    Copy a rectangular area of the given offscreen buffer into the current drawing destination.
- Fl_Offscreen fl_create_offscreen (int w, int h)

    Creation of an offscreen graphics buffer.
- void fl_delete_offscreen (Fl_Offscreen ctx)

    Deletion of an offscreen graphics buffer.

- void fl_end_offscreen ()

  Quit sending drawing commands to the current offscreen buffer.

Variables

- const int **stack_max** = 16

32.14.1 Detailed Description

Fl_Double_Window implementation.

32.15 fl_draw.H File Reference

utility header to pull drawing functions together

```
   #include <FL/x.H>
#include <FL/Enumerations.H>
#include <FL/Fl_Window.H>
#include <FL/Fl_Device.H>
```

Macros

- #define fl_clip fl_push_clip

  Intersects the current clip region with a rectangle and pushes this new region onto the stack (deprecated).

Enumerations

- enum {
  FL_SOLID = 0, FL_DASH = 1, FL_DOT = 2, FL_DASHDOT = 3,
  FL_DASHDOTDOT = 4, FL_CAP_FLAT = 0x100, FL_CAP_ROUND = 0x200, FL_CAP_SQUARE = 0x300,
  FL_JOIN_MITER = 0x1000, FL_JOIN_ROUND = 0x2000, FL_JOIN_BEVEL = 0x3000 }

Functions

- FL_EXPORT int fl_add_symbol (const char ∗name, void(∗drawit)(Fl_Color), int scalable)

  Adds a symbol to the system.
- void fl_arc (int x, int y, int w, int h, double a1, double a2)

  Draw ellipse sections using integer coordinates.
- void fl_arc (double x, double y, double r, double start, double end)

  Adds a series of points to the current path on the arc of a circle.
- void fl_begin_complex_polygon ()

  Starts drawing a complex filled polygon.
- void fl_begin_line ()

  Starts drawing a list of lines.
- void fl_begin_loop ()

  Starts drawing a closed sequence of lines.
- void fl_begin_points ()

  Starts drawing a list of points.
- void fl_begin_polygon ()

Starts drawing a convex filled polygon.
- FL_EXPORT char fl_can_do_alpha_blending ()

  Checks whether platform supports true alpha blending for RGBA images.
- FL_EXPORT void fl_chord (int x, int y, int w, int h, double a1, double a2)

  fl_chord declaration is a place holder - the function does not yet exist
- void fl_circle (double x, double y, double r)

  fl_circle() is equivalent to fl_arc(x,y,r,0,360), but may be faster.
- int fl_clip_box (int x, int y, int w, int h, int &X, int &Y, int &W, int &H)

  Intersects the rectangle with the current clip region and returns the bounding box of the result.
- void fl_clip_region (Fl_Region r)

  Replaces the top of the clipping stack with a clipping region of any shape.
- Fl_Region fl_clip_region ()

  Returns the current clipping region.
- void fl_color (Fl_Color c)

  Sets the color for all subsequent drawing operations.
- void fl_color (int c)

  for back compatibility - use fl_color(Fl_Color c) instead
- void fl_color (uchar r, uchar g, uchar b)

  Sets the color for all subsequent drawing operations.
- Fl_Color fl_color ()

  Returns the last fl_color() that was set.
- FL_EXPORT void fl_cursor (Fl_Cursor)

  Sets the cursor for the current window to the specified shape and colors.
- FL_EXPORT void **fl_cursor** (Fl_Cursor, Fl_Color fg, Fl_Color bg=FL_WHITE)
- void fl_curve (double X0, double Y0, double X1, double Y1, double X2, double Y2, double X3, double Y3)

  Adds a series of points on a Bezier curve to the path.
- int fl_descent ()

  Returns the recommended distance above the bottom of a fl_height() tall box to draw the text at so it looks centered vertically in that box.
- FL_EXPORT void fl_draw (const char ∗str, int x, int y)

  Draws a nul-terminated UTF-8 string starting at the given `x, y` location.
- FL_EXPORT void fl_draw (int angle, const char ∗str, int x, int y)

  Draws a nul-terminated UTF-8 string starting at the given `x, y` location and rotating `angle` degrees counter-clockwise.
- void fl_draw (const char ∗str, int n, int x, int y)

  Draws starting at the given `x, y` location a UTF-8 string of length `n` bytes.
- void fl_draw (int angle, const char ∗str, int n, int x, int y)

  Draws at the given `x, y` location a UTF-8 string of length `n` bytes rotating `angle` degrees counter-clockwise.
- FL_EXPORT void fl_draw (const char ∗str, int x, int y, int w, int h, Fl_Align align, Fl_Image ∗img=0, int draw_symbols=1)

  Fancy string drawing function which is used to draw all the labels.
- FL_EXPORT void fl_draw (const char ∗str, int x, int y, int w, int h, Fl_Align align, void(∗callthis)(const char ∗, int, int, int), Fl_Image ∗img=0, int draw_symbols=1)

  The same as fl_draw(const char∗,int,int,int,int,Fl_Align,Fl_Image∗,int) with the addition of the `callthis` parameter, which is a pointer to a text drawing function such as fl_draw(const char∗, int, int, int) to do the real work.

- FL_EXPORT void fl_draw_box (Fl_Boxtype, int x, int y, int w, int h, Fl_Color)

  Draws a box using given type, position, size and color.
- void fl_draw_image (const uchar ∗buf, int X, int Y, int W, int H, int D=3, int L=0)

  Draws an 8-bit per color RGB or luminance image.
- void fl_draw_image (Fl_Draw_Image_Cb cb, void ∗data, int X, int Y, int W, int H, int D=3)

  Draws an image using a callback function to generate image data.
- void fl_draw_image_mono (const uchar ∗buf, int X, int Y, int W, int H, int D=1, int L=0)

  Draws a gray-scale (1 channel) image.
- void fl_draw_image_mono (Fl_Draw_Image_Cb cb, void ∗data, int X, int Y, int W, int H, int D=1)

  Draws a gray-scale image using a callback function to generate image data.
- FL_EXPORT int fl_draw_pixmap (char ∗const ∗data, int x, int y, Fl_Color=FL_GRAY)

  Draw XPM image data, with the top-left corner at the given position.
- FL_EXPORT int fl_draw_pixmap (const char ∗const ∗cdata, int x, int y, Fl_Color=FL_GRAY)

  Draw XPM image data, with the top-left corner at the given position.
- FL_EXPORT int fl_draw_symbol (const char ∗label, int x, int y, int w, int h, Fl_Color)

  Draw the named symbol in the given rectangle using the given color.
- void fl_end_complex_polygon ()

  Ends complex filled polygon, and draws.
- void fl_end_line ()

  Ends list of lines, and draws.
- void fl_end_loop ()

  Ends closed sequence of lines, and draws.
- void fl_end_points ()

  Ends list of points, and draws.
- void fl_end_polygon ()

  Ends convex filled polygon, and draws.
- FL_EXPORT const char ∗ fl_expand_text (const char ∗from, char ∗buf, int maxbuf, double maxw, int &n, double &width, int wrap, int draw_symbols=0)

  Copy `from` to `buf`, replacing control characters with ^X.
- void fl_font (Fl_Font face, Fl_Fontsize fsize)

  Sets the current font, which is then used in various drawing routines.
- Fl_Font fl_font ()

  Returns the `face` set by the most recent call to fl_font().
- FL_EXPORT void fl_frame (const char ∗s, int x, int y, int w, int h)

  Draws a series of line segments around the given box.
- FL_EXPORT void fl_frame2 (const char ∗s, int x, int y, int w, int h)

  Draws a series of line segments around the given box.
- void fl_gap ()

  Call fl_gap() to separate loops of the path.
- int fl_height ()

  Returns the recommended minimum line spacing for the current font.
- FL_EXPORT int fl_height (int font, int size)

  This function returns the actual height of the specified `font` and `size`.
- FL_EXPORT const char ∗ fl_latin1_to_local (const char ∗t, int n=-1)

  Converts text from Windows/X11 latin1 character set to local encoding.
- void fl_line (int x, int y, int x1, int y1)

  Draws a line from (x,y) to (x1,y1)

- void fl_line (int x, int y, int x1, int y1, int x2, int y2)

  Draws a line from (x,y) to (x1,y1) and another from (x1,y1) to (x2,y2)
- void fl_line_style (int style, int width=0, char ∗dashes=0)

  Sets how to draw lines (the "pen").
- FL_EXPORT const char ∗ fl_local_to_latin1 (const char ∗t, int n=-1)

  Converts text from local encoding to Windowx/X11 latin1 character set.
- FL_EXPORT const char ∗ fl_local_to_mac_roman (const char ∗t, int n=-1)

  Converts text from local encoding to Mac Roman character set.
- void fl_loop (int x, int y, int x1, int y1, int x2, int y2)

  Outlines a 3-sided polygon with lines.
- void fl_loop (int x, int y, int x1, int y1, int x2, int y2, int x3, int y3)

  Outlines a 4-sided polygon with lines.
- FL_EXPORT const char ∗ fl_mac_roman_to_local (const char ∗t, int n=-1)

  Converts text from Mac Roman character set to local encoding.
- FL_EXPORT void fl_measure (const char ∗str, int &x, int &y, int draw_symbols=1)

  Measure how wide and tall the string will be when printed by the fl_draw() function with `align` parameter.
- FL_EXPORT int fl_measure_pixmap (char ∗const ∗data, int &w, int &h)

  Get the dimensions of a pixmap.
- FL_EXPORT int fl_measure_pixmap (const char ∗const ∗cdata, int &w, int &h)

  Get the dimensions of a pixmap.
- void fl_mult_matrix (double a, double b, double c, double d, double x, double y)

  Concatenates another transformation onto the current one.
- int fl_not_clipped (int x, int y, int w, int h)

  Does the rectangle intersect the current clip region?
- FL_EXPORT unsigned int fl_old_shortcut (const char ∗s)

  Emulation of XForms named shortcuts.
- FL_EXPORT void fl_overlay_clear ()

  Erase a selection rectangle without drawing a new one.
- FL_EXPORT void fl_overlay_rect (int x, int y, int w, int h)

  Draws a selection rectangle, erasing a previous one by XOR'ing it first.
- void fl_pie (int x, int y, int w, int h, double a1, double a2)

  Draw filled ellipse sections using integer coordinates.
- void fl_point (int x, int y)

  Draws a single pixel at the given coordinates.
- void fl_polygon (int x, int y, int x1, int y1, int x2, int y2)

  Fills a 3-sided polygon.
- void fl_polygon (int x, int y, int x1, int y1, int x2, int y2, int x3, int y3)

  Fills a 4-sided polygon.
- void fl_pop_clip ()

  Restores the previous clip region.
- void fl_pop_matrix ()

  Restores the current transformation matrix from the stack.
- void fl_push_clip (int x, int y, int w, int h)

  Intersects the current clip region with a rectangle and pushes this new region onto the stack.
- void fl_push_matrix ()

  Saves the current transformation matrix on the stack.

- void fl_push_no_clip ()

  Pushes an empty clip region onto the stack so nothing will be clipped.
- FL_EXPORT uchar ∗ fl_read_image (uchar ∗p, int X, int Y, int W, int H, int alpha=0)

  Reads an RGB(A) image from the current window or off-screen buffer.
- void fl_rect (int x, int y, int w, int h)

  Draws a 1-pixel border inside the given bounding box.
- void fl_rect (int x, int y, int w, int h, Fl_Color c)

  Draws with passed color a 1-pixel border inside the given bounding box.
- void fl_rectf (int x, int y, int w, int h)

  Colors with current color a rectangle that exactly fills the given bounding box.
- void fl_rectf (int x, int y, int w, int h, Fl_Color c)

  Colors with passed color a rectangle that exactly fills the given bounding box.
- FL_EXPORT void fl_rectf (int x, int y, int w, int h, uchar r, uchar g, uchar b)

  Colors a rectangle with "exactly" the passed `r,g,b` color.
- FL_EXPORT void fl_reset_spot (void)
- void fl_restore_clip ()

  Undoes any clobbering of clip done by your program.
- void fl_rotate (double d)

  Concatenates rotation transformation onto the current one.
- void fl_rtl_draw (const char ∗str, int n, int x, int y)

  Draws a UTF-8 string of length `n` bytes right to left starting at the given `x`, `y` location.
- void fl_scale (double x, double y)

  Concatenates scaling transformation onto the current one.
- void fl_scale (double x)

  Concatenates scaling transformation onto the current one.
- FL_EXPORT void fl_scroll (int X, int Y, int W, int H, int dx, int dy, void(∗draw_area)(void ∗, int, int, int, int), void ∗data)

  Scroll a rectangle and draw the newly exposed portions.
- FL_EXPORT void fl_set_spot (int font, int size, int X, int Y, int W, int H, Fl_Window ∗win=0)
- FL_EXPORT void fl_set_status (int X, int Y, int W, int H)
- FL_EXPORT const char ∗ fl_shortcut_label (unsigned int shortcut)

  Get a human-readable string from a shortcut value.
- FL_EXPORT const char ∗ fl_shortcut_label (unsigned int shortcut, const char ∗∗eom)

  Get a human-readable string from a shortcut value.
- Fl_Fontsize fl_size ()

  Returns the `size` set by the most recent call to fl_font().
- FL_EXPORT void fl_text_extents (const char ∗, int &dx, int &dy, int &w, int &h)

  Determines the minimum pixel dimensions of a nul-terminated string.
- void fl_text_extents (const char ∗t, int n, int &dx, int &dy, int &w, int &h)

  Determines the minimum pixel dimensions of a sequence of `n` characters.
- double fl_transform_dx (double x, double y)

  Transforms distance using current transformation matrix.
- double fl_transform_dy (double x, double y)

  Transforms distance using current transformation matrix.
- double fl_transform_x (double x, double y)

  Transforms coordinate using the current transformation matrix.
- double fl_transform_y (double x, double y)

Transforms coordinate using the current transformation matrix.
- void fl_transformed_vertex (double xf, double yf)

  Adds coordinate pair to the vertex list without further transformations.
- void fl_translate (double x, double y)

  Concatenates translation transformation onto the current one.
- void fl_vertex (double x, double y)

  Adds a single vertex to the current path.
- FL_EXPORT double fl_width (const char ∗txt)

  Returns the typographical width of a nul-terminated string using the current font face and size.
- double fl_width (const char ∗txt, int n)

  Returns the typographical width of a sequence of `n` *characters using the current font face and size.*
- double fl_width (unsigned int c)

  Returns the typographical width of a single character using the current font face and size.
- void fl_xyline (int x, int y, int x1)

  Draws a horizontal line from (x,y) to (x1,y)
- void fl_xyline (int x, int y, int x1, int y2)

  Draws a horizontal line from (x,y) to (x1,y), then vertical from (x1,y) to (x1,y2)
- void fl_xyline (int x, int y, int x1, int y2, int x3)

  Draws a horizontal line from (x,y) to (x1,y), then a vertical from (x1,y) to (x1,y2) and then another horizontal from (x1,y2) to (x3,y2)
- void fl_yxline (int x, int y, int y1)

  Draws a vertical line from (x,y) to (x,y1)
- void fl_yxline (int x, int y, int y1, int x2)

  Draws a vertical line from (x,y) to (x,y1), then a horizontal from (x,y1) to (x2,y1)
- void fl_yxline (int x, int y, int y1, int x2, int y3)

  Draws a vertical line from (x,y) to (x,y1) then a horizontal from (x,y1) to (x2,y1), then another vertical from (x2,y1) to (x2,y3)

Variables

- FL_EXPORT char **fl_draw_shortcut**

32.15.1 Detailed Description

utility header to pull drawing functions together

32.16 Fl_Image.H File Reference

Fl_Image, Fl_RGB_Image classes.

```
   #include "Enumerations.H"
#include <stdlib.h>
```

Classes

- class Fl_Image

  Base class for image caching and drawing.
- class Fl_RGB_Image

  The Fl_RGB_Image class supports caching and drawing of full-color images with 1 to 4 channels of color information.

Enumerations

- enum Fl_RGB_Scaling { FL_RGB_SCALING_NEAREST = 0, FL_RGB_SCALING_BILINEAR }

  The scaling algorithm to use for RGB images.

32.16.1 Detailed Description

Fl_Image, Fl_RGB_Image classes.

32.16.2 Enumeration Type Documentation

enum Fl_RGB_Scaling

The scaling algorithm to use for RGB images.

Enumerator

FL_RGB_SCALING_NEAREST	default RGB image scaling algorithm
FL_RGB_SCALING_BILINEAR	more accurate, but slower RGB image scaling algorithm

32.17 fl_line_style.cxx File Reference

Line style drawing utility hiding different platforms.

```
   #include <FL/Fl.H>
#include <FL/fl_draw.H>
#include <FL/x.H>
#include <FL/Fl_Printer.H>
#include "flstring.h"
#include <stdio.h>
```

Variables

- int **fl_line_width_** = 0

32.17.1 Detailed Description

Line style drawing utility hiding different platforms.

32.18 Fl_Menu_Item.H File Reference

```
#include "Fl_Widget.H"
#include "Fl_Image.H"
```

Classes

- struct Fl_Menu_Item

  The Fl_Menu_Item structure defines a single menu item that is used by the Fl_Menu_ class.

Typedefs

- typedef Fl_Menu_Item **Fl_Menu**

Enumerations

- enum {
  FL_MENU_INACTIVE = 1, FL_MENU_TOGGLE = 2, FL_MENU_VALUE = 4, FL_MENU_RAD-IO = 8,
  FL_MENU_INVISIBLE = 0x10, FL_SUBMENU_POINTER = 0x20, FL_SUBMENU = 0x40, FL_-MENU_DIVIDER = 0x80,
  FL_MENU_HORIZONTAL = 0x100 }
- enum {
  FL_PUP_NONE = 0, **FL_PUP_GREY** = FL_MENU_INACTIVE, **FL_PUP_GRAY** = FL_MENU_-INACTIVE, **FL_MENU_BOX** = FL_MENU_TOGGLE,
  FL_PUP_BOX = FL_MENU_TOGGLE, **FL_MENU_CHECK** = FL_MENU_VALUE, **FL_PUP_C-HECK** = FL_MENU_VALUE, **FL_PUP_RADIO** = FL_MENU_RADIO,
  FL_PUP_INVISIBLE = FL_MENU_INVISIBLE, **FL_PUP_SUBMENU** = FL_SUBMENU_POIN-TER }

Functions

- FL_EXPORT Fl_Shortcut fl_old_shortcut (const char ∗)

  Emulation of XForms named shortcuts.

32.18.1 Enumeration Type Documentation

anonymous enum

Enumerator

FL_MENU_INACTIVE Deactivate menu item (gray out)

FL_MENU_TOGGLE Item is a checkbox toggle (shows checkbox for on/off state)

FL_MENU_VALUE The on/off state for checkbox/radio buttons (if set, state is ’on’)

FL_MENU_RADIO Item is a radio button (one checkbox of many can be on)

FL_MENU_INVISIBLE Item will not show up (shortcut will work)

FL_SUBMENU_POINTER Indicates user_data() is a pointer to another menu array.

FL_SUBMENU This item is a submenu to other items.

FL_MENU_DIVIDER Creates divider line below this item. Also ends a group of radio buttons.

FL_MENU_HORIZONTAL ??? – reserved

32.19 Fl_Native_File_Chooser.H File Reference

Fl_Native_File_Chooser widget.

```
#include <FL/Fl_File_Chooser.H>
```

Classes

- class Fl_FLTK_File_Chooser
- class Fl_GTK_File_Chooser
- class Fl_Native_File_Chooser

  This class lets an FLTK application easily and consistently access the operating system’s native file chooser.

32.19.1 Detailed Description

Fl_Native_File_Chooser widget.

32.20 Fl_Paged_Device.cxx File Reference

implementation of class Fl_Paged_Device.

```
   #include <FL/Fl_Paged_Device.H>
#include <FL/Fl.H>
#include <FL/fl_draw.H>
```

32.20.1 Detailed Description

implementation of class Fl_Paged_Device.

32.21 Fl_Paged_Device.H File Reference

declaration of class Fl_Paged_Device.

```
   #include <FL/Fl_Device.H>
#include <FL/Fl_Window.H>
```

Classes

- class Fl_Paged_Device

  Represents page-structured drawing surfaces.
- struct Fl_Paged_Device::page_format

  width, height and name of a page format

Macros

- #define NO_PAGE_FORMATS 30 /∗ MSVC6 compilation fix ∗/

  Number of elements in enum Page_Format.

32.21.1 Detailed Description

declaration of class Fl_Paged_Device.

32.22 Fl_PostScript.H File Reference

declaration of classes Fl_PostScript_Graphics_Driver, Fl_PostScript_File_Device.

```
   #include <FL/Fl_Paged_Device.H>
#include <FL/fl_draw.H>
#include <stdarg.h>
```

Classes

- class Fl_PostScript_File_Device

  To send graphical output to a PostScript file.
- class Fl_PostScript_Graphics_Driver

  PostScript graphical backend.

Typedefs

- typedef int(**Fl_PostScript_Close_Command**)(FILE ∗)

32.22.1 Detailed Description

declaration of classes Fl_PostScript_Graphics_Driver, Fl_PostScript_File_Device.

32.23 Fl_Printer.H File Reference

declaration of classes Fl_Printer, Fl_System_Printer and Fl_PostScript_Printer.

```
   #include <FL/x.H>
#include <FL/Fl_Paged_Device.H>
#include <FL/fl_draw.H>
#include <FL/Fl_Pixmap.H>
#include <FL/Fl_RGB_Image.H>
#include <FL/Fl_Bitmap.H>
#include <stdio.h>
#include <FL/Fl_PostScript.H>
```

Classes

- class Fl_PostScript_Printer

  Print support under Unix/Linux.
- class Fl_Printer

  OS-independent print support.
- class Fl_System_Printer

  Print support under MSWindows and Mac OS.

32.23.1 Detailed Description

declaration of classes Fl_Printer, Fl_System_Printer and Fl_PostScript_Printer.

32.24 fl_rect.cxx File Reference

Drawing and clipping routines for rectangles.

```
   #include <config.h>
#include <FL/Fl.H>
#include <FL/Fl_Widget.H>
#include <FL/Fl_Printer.H>
#include <FL/fl_draw.H>
#include <FL/x.H>
```

Functions

- Fl_Region **XRectangleRegion** (int x, int y, int w, int h)

Variables

- int **fl_line_width_**

32.24.1 Detailed Description

Drawing and clipping routines for rectangles.

32.25 Fl_Shared_Image.H File Reference

Fl_Shared_Image class.

```
#include "Fl_Image.H"
```

Classes

- class Fl_Shared_Image

    This class supports caching, loading, scaling, and drawing of image files.

Typedefs

- typedef Fl_Image ∗(∗ **Fl_Shared_Handler**)(const char ∗name, uchar ∗header, int headerlen)

Functions

- FL_EXPORT void fl_register_images ()

    Register the image formats.

32.25.1 Detailed Description

Fl_Shared_Image class.

32.25.2 Function Documentation

FL_EXPORT void fl_register_images ( )

Register the image formats.

This function is provided in the fltk_images library and registers all of the "extra" image file formats that are not part of the core FLTK library.

32.26 fl_show_colormap.H File Reference

The fl_show_colormap() function hides the implementation classes used to provide the popup window and color selection mechanism.

Functions

- FL_EXPORT Fl_Color fl_show_colormap (Fl_Color oldcol)

    Pops up a window to let the user pick a colormap entry.

32.26.1 Detailed Description

The fl_show_colormap() function hides the implementation classes used to provide the popup window and color selection mechanism.

32.27 Fl_Tree.H File Reference

This file contains the definitions of the Fl_Tree class.

```
   #include <FL/Fl.H>
#include <FL/Fl_Group.H>
#include <FL/Fl_Scrollbar.H>
#include <FL/fl_draw.H>
#include <FL/Fl_Tree_Item.H>
#include <FL/Fl_Tree_Prefs.H>
```

Classes

- class Fl_Tree

  Tree widget.

Enumerations

- enum Fl_Tree_Reason {
  FL_TREE_REASON_NONE =0, FL_TREE_REASON_SELECTED, FL_TREE_REASON_DESELECTED, FL_TREE_REASON_RESELECTED,
  FL_TREE_REASON_OPENED, FL_TREE_REASON_CLOSED, FL_TREE_REASON_DRAGGED
  }

  The reason the callback was invoked.

32.27.1 Detailed Description

This file contains the definitions of the Fl_Tree class.

32.27.2 Enumeration Type Documentation

enum Fl_Tree_Reason

The reason the callback was invoked.

Enumerator

FL_TREE_REASON_NONE unknown reason

FL_TREE_REASON_SELECTED an item was selected

FL_TREE_REASON_DESELECTED an item was de-selected

FL_TREE_REASON_RESELECTED an item was re-selected (e.g. double-clicked)

FL_TREE_REASON_OPENED an item was opened

FL_TREE_REASON_CLOSED an item was closed

FL_TREE_REASON_DRAGGED an item was dragged into a new place

32.28 Fl_Tree_Item.H File Reference

This file contains the definitions for Fl_Tree_Item.

```
   #include <FL/Fl.H>
#include <FL/Fl_Widget.H>
#include <FL/Fl_Image.H>
#include <FL/fl_draw.H>
#include <FL/Fl_Tree_Item_Array.H>
#include <FL/Fl_Tree_Prefs.H>
```

Classes

- class Fl_Tree_Item

  Tree widget item.

32.28.1 Detailed Description

This file contains the definitions for Fl_Tree_Item.

32.29 Fl_Tree_Item_Array.H File Reference

This file defines a class that manages an array of Fl_Tree_Item pointers.

```
   #include <FL/Fl.H>
#include "Fl_Export.H"
```

Classes

- class Fl_Tree_Item_Array

  Manages an array of Fl_Tree_Item pointers.

32.29.1 Detailed Description

This file defines a class that manages an array of Fl_Tree_Item pointers.

32.30 Fl_Tree_Prefs.H File Reference

This file contains the definitions for Fl_Tree's preferences.

```
   #include <FL/Fl.H>
```

Classes

- class Fl_Tree_Prefs

  Tree widget's preferences.

Typedefs

- typedef void(**Fl_Tree_Item_Draw_Callback**)(Fl_Tree_Item ∗, void ∗)

Enumerations

- enum Fl_Tree_Connector { FL_TREE_CONNECTOR_NONE =0, FL_TREE_CONNECTOR_DOTTED =1, FL_TREE_CONNECTOR_SOLID =2 }

  Defines the style of connection lines between items.
- enum Fl_Tree_Item_Draw_Mode { FL_TREE_ITEM_DRAW_DEFAULT =0, FL_TREE_ITEM_DRAW_LABEL_AND_WIDGET =1, FL_TREE_ITEM_HEIGHT_FROM_WIDGET =2 }

  Bit flags that control how item's labels and widget()s are drawn in the tree via item_draw_mode().
- enum Fl_Tree_Item_Reselect_Mode { FL_TREE_SELECTABLE_ONCE =0, FL_TREE_SELECTABLE_ALWAYS }

  Defines the ways an item can be (re) selected via item_reselect_mode().
- enum Fl_Tree_Select { FL_TREE_SELECT_NONE =0, FL_TREE_SELECT_SINGLE =1, FL_TREE_SELECT_MULTI =2, FL_TREE_SELECT_SINGLE_DRAGGABLE =3 }

  Tree selection style.
- enum Fl_Tree_Sort { FL_TREE_SORT_NONE =0, FL_TREE_SORT_ASCENDING =1, FL_TREE_SORT_DESCENDING =2 }

  Sort order options for items added to the tree.

32.30.1 Detailed Description

This file contains the definitions for Fl_Tree's preferences.

```
    Fl_Tree_Prefs
         :
    .....:.......
    :           :
Fl_Tree         :
    |_____ Fl_Tree_Item
```

32.30.2 Enumeration Type Documentation

enum Fl_Tree_Connector

Defines the style of connection lines between items.

Enumerator

FL_TREE_CONNECTOR_NONE Use no lines connecting items.

FL_TREE_CONNECTOR_DOTTED Use dotted lines connecting items (default)

FL_TREE_CONNECTOR_SOLID Use solid lines connecting items.

enum Fl_Tree_Item_Draw_Mode

Bit flags that control how item's labels and widget()s are drawn in the tree via item_draw_mode().

Enumerator

FL_TREE_ITEM_DRAW_DEFAULT If widget() defined, draw in place of label, and widget() tracks item height (default)

FL_TREE_ITEM_DRAW_LABEL_AND_WIDGET If widget() defined, include label to the left of the widget.

FL_TREE_ITEM_HEIGHT_FROM_WIDGET If widget() defined, widget()'s height controls item's height.

enum Fl_Tree_Item_Reselect_Mode

Defines the ways an item can be (re) selected via item_reselect_mode().

Enumerator

FL_TREE_SELECTABLE_ONCE Item can only be selected once (default)

FL_TREE_SELECTABLE_ALWAYS Enables FL_TREE_REASON_RESELECTED events for callbacks.

enum Fl_Tree_Select

Tree selection style.

Enumerator

FL_TREE_SELECT_NONE Nothing selected when items are clicked.

FL_TREE_SELECT_SINGLE Single item selected when item is clicked (default)

FL_TREE_SELECT_MULTI Multiple items can be selected by clicking with SHIFT, CTRL or mouse drags.

FL_TREE_SELECT_SINGLE_DRAGGABLE Single items may be selected, and they may be. reordered by mouse drag.

enum Fl_Tree_Sort

Sort order options for items added to the tree.

Enumerator

FL_TREE_SORT_NONE No sorting; items are added in the order defined (default).

FL_TREE_SORT_ASCENDING Add items in ascending sort order.

FL_TREE_SORT_DESCENDING Add items in descending sort order.

32.31 fl_types.h File Reference

This file contains simple "C"-style type definitions.

Typedefs

Miscellaneous

- typedef unsigned char uchar

  unsigned char
- typedef unsigned long ulong

  unsigned long
- typedef char ∗ Fl_String

  Flexible length UTF-8 Unicode text.
- typedef const char ∗ Fl_CString

  Flexible length UTF-8 Unicode read-only string.
- typedef unsigned int Fl_Shortcut

  24-bit Unicode character + 8-bit indicator for keyboard flags
- typedef unsigned int Fl_Char

  24-bit Unicode character - upper 8 bits are unused

32.31.1 Detailed Description

This file contains simple "C"-style type definitions.

32.31.2 Typedef Documentation

typedef const char∗ Fl_CString

Flexible length UTF-8 Unicode read-only string.

See Also

Fl_String

typedef char∗ Fl_String

Flexible length UTF-8 Unicode text.

Todo FIXME: temporary (?) typedef to mark UTF-8 and Unicode conversions

32.32 fl_utf8.h File Reference

header for Unicode and UTF-8 character handling

```
   #include "Fl_Export.H"
#include "fl_types.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <locale.h>
```

Macros

- #define **xchar** unsigned short

Functions

- FL_EXPORT int fl_access (const char ∗f, int mode)

  Cross-platform function to test a files access() with a UTF-8 encoded name or value.
- FL_EXPORT int fl_chmod (const char ∗f, int mode)

  Cross-platform function to set a files mode() with a UTF-8 encoded name or value.
- FL_EXPORT int **fl_execvp** (const char ∗file, char ∗const ∗argv)
- FL_EXPORT FILE ∗ fl_fopen (const char ∗f, const char ∗mode)

  Cross-platform function to open files with a UTF-8 encoded name.
- FL_EXPORT char ∗ fl_getcwd (char ∗b, int l)

  Cross-platform function to get the current working directory as a UTF-8 encoded value.
- FL_EXPORT char ∗ fl_getenv (const char ∗v)

  Cross-platform function to get environment variables with a UTF-8 encoded name or value.
- FL_EXPORT char fl_make_path (const char ∗path)

  Cross-platform function to recursively create a path in the file system.
- FL_EXPORT void fl_make_path_for_file (const char ∗path)

Cross-platform function to create a path for the file in the file system.

- FL_EXPORT int fl_mkdir (const char ∗f, int mode)

  Cross-platform function to create a directory with a UTF-8 encoded name.
- FL_EXPORT unsigned int fl_nonspacing (unsigned int ucs)

  Returns true if the Unicode character `ucs` is non-spacing.
- FL_EXPORT int fl_open (const char ∗f, int oflags,...)

  Cross-platform function to open files with a UTF-8 encoded name.
- FL_EXPORT int fl_rename (const char ∗f, const char ∗n)

  Cross-platform function to rename a filesystem object using UTF-8 encoded names.
- FL_EXPORT int fl_rmdir (const char ∗f)

  Cross-platform function to remove a directory with a UTF-8 encoded name.
- FL_EXPORT int fl_stat (const char ∗f, struct stat ∗b)

  Cross-platform function to stat() a file using a UTF-8 encoded name or value.
- FL_EXPORT int fl_system (const char ∗cmd)

  Cross-platform function to run a system command with a UTF-8 encoded string.
- FL_EXPORT int fl_tolower (unsigned int ucs)

  Returns the Unicode lower case value of `ucs`.
- FL_EXPORT int fl_toupper (unsigned int ucs)

  Returns the Unicode upper case value of `ucs`.
- FL_EXPORT unsigned fl_ucs_to_Utf16 (const unsigned ucs, unsigned short ∗dst, const unsigned dstlen)
- FL_EXPORT int fl_unlink (const char ∗f)

  Cross-platform function to unlink() (that is, delete) a file using a UTF-8 encoded filename.
- FL_EXPORT char ∗ fl_utf2mbcs (const char ∗s)

  Converts UTF-8 string `s` to a local multi-byte character string.
- FL_EXPORT const char ∗ fl_utf8back (const char ∗p, const char ∗start, const char ∗end)
- FL_EXPORT int fl_utf8bytes (unsigned ucs)

  Return the number of bytes needed to encode the given UCS4 character in UTF-8.
- FL_EXPORT unsigned fl_utf8decode (const char ∗p, const char ∗end, int ∗len)
- FL_EXPORT int fl_utf8encode (unsigned ucs, char ∗buf)
- FL_EXPORT unsigned fl_utf8from_mb (char ∗dst, unsigned dstlen, const char ∗src, unsigned srclen)
- FL_EXPORT unsigned fl_utf8froma (char ∗dst, unsigned dstlen, const char ∗src, unsigned srclen)
- FL_EXPORT unsigned fl_utf8fromwc (char ∗dst, unsigned dstlen, const wchar_t ∗src, unsigned srclen)
- FL_EXPORT const char ∗ fl_utf8fwd (const char ∗p, const char ∗start, const char ∗end)
- FL_EXPORT int fl_utf8len (char c)

  Returns the byte length of the UTF-8 sequence with first byte `c`, or -1 if `c` is not valid.
- FL_EXPORT int fl_utf8len1 (char c)

  Returns the byte length of the UTF-8 sequence with first byte `c`, or 1 if `c` is not valid.
- FL_EXPORT int fl_utf8locale (void)
- FL_EXPORT int fl_utf8test (const char ∗src, unsigned len)
- FL_EXPORT unsigned fl_utf8to_mb (const char ∗src, unsigned srclen, char ∗dst, unsigned dstlen)
- FL_EXPORT unsigned fl_utf8toa (const char ∗src, unsigned srclen, char ∗dst, unsigned dstlen)
- FL_EXPORT unsigned fl_utf8toUtf16 (const char ∗src, unsigned srclen, unsigned short ∗dst, unsigned dstlen)
- FL_EXPORT unsigned fl_utf8towc (const char ∗src, unsigned srclen, wchar_t ∗dst, unsigned dstlen)

  Converts a UTF-8 string into a wide character string.
- FL_EXPORT int fl_utf_nb_char (const unsigned char ∗buf, int len)

Returns the number of Unicode chars in the UTF-8 string.

- FL_EXPORT int fl_utf_strcasecmp (const char ∗s1, const char ∗s2)

  UTF-8 aware strcasecmp - converts to Unicode and tests.
- FL_EXPORT int fl_utf_strncasecmp (const char ∗s1, const char ∗s2, int n)

  UTF-8 aware strncasecmp - converts to lower case Unicode and tests.
- FL_EXPORT int fl_utf_tolower (const unsigned char ∗str, int len, char ∗buf)

  Converts the string `str` to its lower case equivalent into buf.
- FL_EXPORT int fl_utf_toupper (const unsigned char ∗str, int len, char ∗buf)

  Converts the string `str` to its upper case equivalent into buf.
- FL_EXPORT int fl_wcwidth (const char ∗src)

  extended wrapper around fl_wcwidth_(unsigned int ucs) function.
- FL_EXPORT int fl_wcwidth_ (unsigned int ucs)

  wrapper to adapt Markus Kuhn's implementation of wcwidth() for FLTK

32.32.1 Detailed Description

header for Unicode and UTF-8 character handling

32.33 fl_vertex.cxx File Reference

Portable drawing code for drawing arbitrary shapes with simple 2D transformations.

```
   #include <config.h>
#include <FL/fl_draw.H>
#include <FL/x.H>
#include <FL/Fl.H>
#include <FL/math.h>
#include <stdlib.h>
```

32.33.1 Detailed Description

Portable drawing code for drawing arbitrary shapes with simple 2D transformations.

32.34 Fl_Widget.H File Reference

Fl_Widget, Fl_Label classes .

```
   #include "Enumerations.H"
```

Classes

- struct Fl_Label

  This struct stores all information for a text or mixed graphics label.
- class Fl_Widget

  Fl_Widget is the base class for all widgets in FLTK.

Macros

- #define FL_RESERVED_TYPE 100

  Reserved type numbers (necessary for my cheapo RTTI) start here.

Typedefs

- typedef void(Fl_Callback)(Fl_Widget ∗, void ∗)

  Default callback type definition for all fltk widgets (by far the most used)
- typedef void(Fl_Callback0)(Fl_Widget ∗)

  One parameter callback type definition passing only the widget.
- typedef void(Fl_Callback1)(Fl_Widget ∗, long)

  Callback type definition passing the widget and a long data value.
- typedef Fl_Callback ∗ Fl_Callback_p

  Default callback type pointer definition for all fltk widgets.
- typedef long fl_intptr_t
- typedef unsigned long **fl_uintptr_t**

32.34.1 Detailed Description

Fl_Widget, Fl_Label classes .

32.34.2 Macro Definition Documentation

#define FL_RESERVED_TYPE 100

Reserved type numbers (necessary for my cheapo RTTI) start here.
Grep the header files for "RESERVED_TYPE" to find the next available number.

32.34.3 Typedef Documentation

typedef long fl_intptr_t

Todo typedef's fl_intptr_t and fl_uintptr_t should be documented.

32.35 Fl_Window.H File Reference

Fl_Window widget .

```
   #include "Fl_Group.H"
#include "Fl_Bitmap.H"
#include <stdlib.h>
```

Classes

- class Fl_Window

  This widget produces an actual window.
- struct Fl_Window::shape_data_type

  Data supporting a non-rectangular window shape.

Macros

- #define FL_DOUBLE_WINDOW 0xF1

  double window type id
- #define FL_WINDOW 0xF0

  window type id all subclasses have type() >= this

32.35.1 Detailed Description

Fl_Window widget .

32.36 gl.h File Reference

This file defines wrapper functions for OpenGL in FLTK.

```
   #include "Enumerations.H"
#include <GL/gl.h>
```

Functions

- FL_EXPORT void gl_color (Fl_Color i)

  Sets the curent OpenGL color to an FLTK color.
- void gl_color (int c)

  back compatibility
- FL_EXPORT int gl_descent ()

  Returns the current font's descent.
- FL_EXPORT void gl_draw (const char ∗)

  Draws a nul-terminated string in the current font at the current position.
- FL_EXPORT void gl_draw (const char ∗, int n)

  Draws an array of n characters of the string in the current font at the current position.
- FL_EXPORT void gl_draw (const char ∗, int x, int y)

  Draws a nul-terminated string in the current font at the given position.
- FL_EXPORT void gl_draw (const char ∗, float x, float y)

  Draws a nul-terminated string in the current font at the given position.
- FL_EXPORT void gl_draw (const char ∗, int n, int x, int y)

  Draws n characters of the string in the current font at the given position.
- FL_EXPORT void gl_draw (const char ∗, int n, float x, float y)

  Draws n characters of the string in the current font at the given position.
- FL_EXPORT void gl_draw (const char ∗, int x, int y, int w, int h, Fl_Align)

  Draws a string formatted into a box, with newlines and tabs expanded, other control characters changed to ^X.
- FL_EXPORT void **gl_draw_image** (const uchar ∗, int x, int y, int w, int h, int d=3, int ld=0)
- FL_EXPORT void gl_finish ()

  Releases an OpenGL context.
- FL_EXPORT void gl_font (int fontid, int size)

  Sets the current OpenGL font to the same font as calling fl_font()
- FL_EXPORT int gl_height ()

  Returns the current font's height.
- FL_EXPORT void gl_measure (const char ∗, int &x, int &y)

  Measure how wide and tall the string will be when drawn by the gl_draw() function.
- FL_EXPORT void gl_rect (int x, int y, int w, int h)

  Outlines the given rectangle with the current color.
- void gl_rectf (int x, int y, int w, int h)

  Fills the given rectangle with the current color.
- FL_EXPORT void gl_start ()

Creates an OpenGL context.

- FL_EXPORT double gl_width (const char ∗)

  Returns the width of the string in the current fnt.
- FL_EXPORT double gl_width (const char ∗, int n)

  Returns the width of n characters of the string in the current font.
- FL_EXPORT double gl_width (uchar)

  Returns the width of the character in the current font.

32.36.1 Detailed Description

This file defines wrapper functions for OpenGL in FLTK. To use OpenGL from within an FLTK application you MUST use gl_visual() to select the default visual before doing show() on any windows. Mesa will crash if yoy try to use a visual not returned by glxChooseVidual.

This does not work with Fl_Double_Window's! It will try to draw into the front buffer. Depending on the system this will either crash or do nothing (when pixmaps are being used as back buffer and GL is being done by hardware), work correctly (when GL is done with software, such as Mesa), or draw into the front buffer and be erased when the buffers are swapped (when double buffer hardware is being used)

32.36.2 Function Documentation

FL_EXPORT void gl_color (Fl_Color *i*)

Sets the curent OpenGL color to an FLTK color.

For color-index modes it will use fl_xpixel(c), which is only right if the window uses the default colormap!

FL_EXPORT void gl_draw (const char ∗ *str*)

Draws a nul-terminated string in the current font at the current position.

See Also

On the Mac OS X platform, see gl_texture_pile_height(int)

FL_EXPORT void gl_draw (const char ∗ *str,* int *n*)

Draws an array of n characters of the string in the current font at the current position.

See Also

On the Mac OS X platform, see gl_texture_pile_height(int)

FL_EXPORT void gl_draw (const char ∗ *str,* int *x,* int *y*)

Draws a nul-terminated string in the current font at the given position.

See Also

On the Mac OS X platform, see gl_texture_pile_height(int)

FL_EXPORT void gl_draw (const char ∗ *str,* float *x,* float *y*)

Draws a nul-terminated string in the current font at the given position.

See Also

On the Mac OS X platform, see gl_texture_pile_height(int)

FL_EXPORT void gl_draw (const char ∗ *str,* int *n,* int *x,* int *y*)

Draws n characters of the string in the current font at the given position.

See Also

On the Mac OS X platform, see gl_texture_pile_height(int)

FL_EXPORT void gl_draw (const char ∗ *str,* int *n,* float *x,* float *y*)

Draws n characters of the string in the current font at the given position.

See Also

On the Mac OS X platform, see gl_texture_pile_height(int)

FL_EXPORT void gl_draw (const char ∗ *str,* int *x,* int *y,* int *w,* int *h,* Fl_Align *align*)

Draws a string formatted into a box, with newlines and tabs expanded, other control characters changed to ^X.

and aligned with the edges or center. Exactly the same output as fl_draw().

FL_EXPORT void gl_rect (int *x,* int *y,* int *w,* int *h*)

Outlines the given rectangle with the current color.

If Fl_Gl_Window::ortho() has been called, then the rectangle will exactly fill the given pixel rectangle.

void gl_rectf (int *x,* int *y,* int *w,* int *h*) `[inline]`

Fills the given rectangle with the current color.

See Also

gl_rect(int x, int y, int w, int h)

32.37 mac.H File Reference

Mac OS X-specific symbols.

Classes

- class Fl_Mac_App_Menu

  Mac OS-specific class allowing to customize and localize the application menu.

Functions

- void fl_mac_set_about (Fl_Callback ∗cb, void ∗user_data, int shortcut=0)

  Attaches a callback to the "About myprog" item of the system application menu.
- void fl_open_callback (void(∗cb)(const char ∗))

  Register a function called for each file dropped onto an application icon.

Variables

- int fl_mac_os_version

  The version number of the running Mac OS X (e.g., 100604 for 10.6.4)
- class Fl_Sys_Menu_Bar ∗ fl_sys_menu_bar

  The system menu bar.

32.37.1 Detailed Description

Mac OS X-specific symbols.

Index