

Configuration Management Plan

Last Updated 11 Apr 2012

Contents

- [Introduction](#)
- [Communication](#)
- [Specific Goals](#)
- [Software Development Practices](#)
 - ◆ [Documentation](#)
 - ◆ [Build System](#)
 - ◆ [Packaging](#)
 - ◆ [FLTK Release Process](#)
 - ◇ [Creating the Release Archives](#)
 - ◇ [Testing the Archives on Linux](#)
 - ◇ [Testing the Archives on Mac OS X](#)
 - ◇ [Testing the Archives on Windows](#)
 - ◇ [Uploading the Files to the Web Server](#)
- [Trouble Report Processing](#)
- [Software Releases](#)
 - ◆ [Version Numbering](#)
 - ◆ [Generation](#)
 - ◆ [Testing](#)
 - ◆ [Releases](#)
 - ◇ [Beta Releases](#)
 - ◇ [Release Candidates](#)
 - ◇ [Production Releases](#)
 - ◇ [Major Releases](#)
 - ◇ [Minor Releases](#)
 - ◇ [Patch Releases](#)
- [File Management](#)
 - [Configuration Management](#)
 - [Directory Structure](#)
 - [Source Files](#)
 - [Header Files](#)
 - [Makefiles](#)
- [Coding Standards](#)
 - [General Coding Style](#)
 - [Source File Documentation](#)
 - [Documenation \(Comments\)](#)
 - [General Developer Recommendations](#)
 - [Methodology, Algorithms, Etc.](#)
 - [C++ Portability](#)
 - ◆ [FLTK 1.1.x Restrictions](#)
 - ◆ [FLTK 2.0.x Restrictions](#)
 - [Source File Naming](#)
 - [Function/Method/Variable Naming](#)
 - [Structure/Class Naming](#)
 - [Constant/Enumeration Naming](#)
 - [Preprocessor Variables](#)
 - ◆ [WIN32](#)
 - ◆ [CYGWIN](#)
 - ◆ [APPLE](#)
 - ◆ [APPLE OUARTZ](#)
 - ◆ [USE X11](#)
 - ◆ [USE XFT](#)
 - ◆ [FL_LIBRARY](#)
 - ◆ [FL_INTERNALS](#)

Configuration Management Plan - Fast Light Toolkit (FLTK)

- ♦ [FL_DOXYGEN](#)
- ♦ [FLTK_ABI_VERSION](#)
- [Miscellaneous](#)
- [Makefile Standards](#)
 - [General Organization](#)
 - [Makefile Documentation](#)
 - [Portable Makefile Construction](#)
 - [Standard Variables](#)
 - [Standard Targets](#)
 - [Object Files](#)
 - [Programs](#)
 - [Static Libraries](#)
 - [Shared Libraries](#)
 - [Dependencies](#)
 - [Install/Uninstall Support](#)
- [Developer Reference](#)
 - [Comment Style](#)
 - [General Developer Recommendations](#)

Introduction

This document defines the processes and standards that all FLTK developers must follow when developing and documenting FLTK, and how trouble reports are handled and releases are generated. The purpose of defining formal processes and standards is to organize and focus our development efforts, ensure that all developers communicate and develop software with a common vocabulary/style, and make it possible for us to generate and release a high-quality GUI toolkit which can be used with a high degree of confidence.

Much of this file describes the existing practices that have been used up through FLTK 1.1.x, however I have also added some new processes/standards to use for future code and releases.

Communication

The `fltk-dev` mailing list and `fltk.development` newsgroup are the primary means of communication between developers. *All* major design changes must be discussed prior to implementation.

Specific Goals

The specific goals of the FLTK are as follows:

- Develop a C++ GUI toolkit based upon sound object-oriented design principles and experience. (*)
- Minimize CPU usage (fast). (*)
- Minimize memory usage (light). (*)
- Support multiple operating systems and windowing environments, including UNIX/Linux, MacOS X, Microsoft Windows, and X11, using the "native" graphics interfaces. (*)
- Support OpenGL rendering in environments that provide it. (*)
- Provide a graphical development environment for designing GUI interfaces, classes, and simple programs. (*)
- Support UTF-8 text.
- Support printer rendering in environments that provide it.
- Support "schemes", "styles", "themes", "skinning", etc. to alter the appearance of widgets in the toolkit easily and efficiently. The purpose is to allow applications to tailor their appearance to the

Configuration Management Plan - Fast Light Toolkit (FLTK)

underlying OS or based upon personal/user preferences.

- Support newer C++ language features, such as templating via the Standard Template Library ("STL"), and certain Standard C++ library interfaces, such as streams. However, FLTK will not depend upon such features and interfaces to minimize portability issues.
- Support intelligent layout of widgets.

Many of these goals are satisfied by FLTK 1.1.x (*), and many complex applications have been written using FLTK on a wide range of platforms and devices. Development of the remaining features is proceeding for FLTK 2.0 with a new, namespace-based API. While 2.0 offers some limited 1.x source compatibility, the changes to the underlying widget classes are significant enough to prevent full compatibility.

Software Development Practices

Documentation

All widgets are documented using the Doxygen software; Doxygen comments are placed in the header file for the class comments and any inline methods, while non-inline methods should have their comments placed in the corresponding source file. The purpose of this separation is to place the comments near the implementation to reduce the possibility of the documentation getting out of sync with the code.

All widgets must have a corresponding test program which exercises all widget functionality and can be used to generate image(s) for the documentation. Complex widgets must have a written tutorial, either as full text or an outline for later publication

The final manuals are formatted using the HTMLDOC software.

Build System

The FLTK build system uses GNU autoconf to tailor the library to the local operating system. Project files for major IDEs are also provided for Microsoft Windows®. To improve portability, makefiles must not make use of the unique features offered by GNU make. See the Makefile Standards section for a description of the allowed make features and makefile guidelines.

Additional GNU build programs such as GNU automake and GNU libtool must not be used. GNU automake produces non-portable makefiles which depend on GNU-specific extensions, and GNU libtool is not portable or reliable enough for FLTK.

Packaging

Source packages are created using the `makesrcdist` script in the Subversion repository. The script accepts one or two arguments:

```
./makesrcdist version
./makesrcdist snapshot
```

Version should be of the form "1.1.10rc2". "rc2" in this case describes the second release candidate.

The version name "snapshot" creates a snapshot of the Subversion repository without copying the release to the *releases* directory.

Configuration Management Plan - Fast Light Toolkit (FLTK)

Binary packages are not currently distributed by the FLTK team, however the `fltk.spec` and `fltk.list` files may be used to create binary packages on Linux, MacOS X, and UNIX. The `fltk.spec` file produces a binary package using the `rpm` software:

```
rpmbuild -ta fltk-version-source.tar.gz
```

The `fltk.list` file is generated by the `configure` script and produces binary packages for many platforms using the [EPM](#) software. The `portable-dist` and `native-dist` targets of the top-level makefile create portable and native packages, respectively:

```
make portable-dist
make native-dist
```

Future releases of FLTK may include files for use with Microsoft Visual Installer to produce `.msi` files for installation on Microsoft Windows®.

FLTK Release Process

FLTK releases are created on a Linux system with the following software installed:

- Infozip 2.3 or higher to create `.zip` archives
- GNU tar 1.12 or higher to create `.tar` archives
- GNU zip 1.2.4 or higher to create `.tar.gz` archives
- Bzip2 1.0.2 or higher to create `.tar.bz2` archives
- Subversion 1.1 or higher to create the release branch

To test the release files, three systems will be required (or a single Intel Mac running Mac OS X, Windows, and Linux):

- A PC running a recent Linux distribution with GCC, RPM, and EPM installed
- A Mac running MacOS X 10.4 or higher with Xcode and EPM installed
- A PC running Microsoft Windows 2000 or higher with Microsoft Visual C++.NET installed

Creating the Release Archives

Each software release provides three archives:

- `fltk-version-source.tar.gz` (3.8MB as of 1.3.0)
- `fltk-version-docs-pdf.tar.gz` (3.3MB as of 1.3.0)
- `fltk-version-docs-html.tar.gz` (3.4MB as of 1.3.0)

The archives contain the source, ide, and build files, the documentation in a single PDF file, and the same documentation in many HTML files. The following steps are performed to create the release archives:

1. Change directories to a current working copy for the FLTK development branch you are releasing, e.g. `"cd fltk-1.3"`.
2. Run `autoconf` to generate the `configure` script.
3. Run `./configure` to generate makefiles.
4. In the documentation directory, run `make dist`.
5. Back in the `fltk` directory, run the `makesrcdist` script with the release version number, e.g.

```
"/makesrcdist 1.3.0".
```

6. Copy the archives that have been created in the */tmp* directory to a more permanent location.

Testing the Archives on Linux

Run the following commands to test the release:

```
rpmbuild -ta fltk-version-source.tar.bz2
tar xvfz fltk-version-source.tar.gz
cd fltk-version
./configure
make all portable-dist native-dist
cd test
./demo
```

Testing the Archives on Mac OS X

Run the following commands to test the release:

```
tar xvfz fltk-version-source.tar.gz
cd fltk-version
./configure
make all portable-dist native-dist
cd test
./demo
```

Testing the Archives on Microsoft Windows

Extract the files from the .tar.gz archive and then open the *ide/VisualC2010/fltk.sln* file in Visual C. Build the *demo* target in both release and debug modes to confirm that the software compiles, and then run the *demo* target to test that each of the demo programs is functioning properly.

Uploading the Files to the Web Server

Go to the following URL to upload the release archives:

<http://www.fltk.org/site/upload.php>

Upload each file and then verify that you can download the archives from the Maryland server. The files should be available from the other download servers within 24 hours.

To change the order in which the file appear on the download page, or to remove outdated files, you need to create an SVN copy of the web interface on your local machine.

1. Go to the web site maintenance page at <http://www.fltk.org/site/index.php>. You must be logged into the web page and have proper permissions.
2. Click *Commit Database Changes...* to get the new download database from the live page into the main repository
3. Update your local web page repository copy and edit the file *data/software.md5* to your needs. Commit to svn.
4. Go back to the web site maintenance page and click *Update the web site....* Verify the Downloads page. Done.

Trouble Report Processing

Software Trouble Reports ("STRs") are submitted every time a user or vendor experiences a problem with or wants a new feature in the FLTK software. Trouble reports are maintained in a database with one of the following states:

1. STR is closed with complete resolution
2. STR is closed without resolution
3. STR is active (waiting for feedback from submitter)
4. STR is pending (additional information available)
5. STR is new

Trouble reports are classified at one of the following levels by the submitter:

1. Request for enhancement
2. Low, e.g. documentation error
3. Moderate, e.g. unable to compile the software
4. High, e.g. key functionality not working
5. Critical, e.g. application crashes

Level 4 and 5 trouble reports are resolved in the next software release. Level 1 to 3 trouble reports are scheduled for resolution in a specific release at the discretion of the release coordinator.

The scope of the problem is also identified as:

1. Specific to a machine
2. Specific to an operating system
3. Applies to all machines and operating systems

New STRs are posted to the `fltk.bugs` forum for priority 2 to 5 and the `fltk.development` forum for priority 1 STRs. FLTK developers then discuss the STR and vote to form consensus as to a proposed resolution.

If a general usage question is posted as a STR, any developer may immediately close the STR without resolution using the canned "support is not available" response which directs users to the `fltk.general` forum.

During discussion, developers propose possible resolutions for the STR and vote to approve them. Each developer posts one vote of +1 (approve), -1 (veto), or 0 (abstain). At least three developers must vote on the proposal, and the total of all votes must be greater than 0. Once consensus is reached, the STR is assigned to a developer that volunteers to resolve the STR.

The assigned developer summarizes the proposed changes in the STR and makes the required changes, attaching a diff of the changes to the STR as needed. The developer then notifies the submitter that the change has been applied and closes the STR when the resolution is confirmed by the submitter or after two weeks, whichever comes first.

If the proposed changes do not resolve the problem, the developer may unassign the STR to continue discussions on the corresponding forum or privately discuss additional modifications with the submitter in order to resolve the STR. When closing the STR, the developer must set the fix version and the first

Subversion revision number which contains the changes.

Software Releases

Version Numbering

FLTK uses a three-part version number separated by periods to represent the major, minor, and patch release numbers:

```
MAJOR.MINOR.PATCH
2.0.0
2.0.1
2.0.2
2.1.0
```

Beta-test releases are identified by appending the letter B to the major and minor release numbers followed by the build number:

```
MAJOR.MINORbBUILD
2.0b1
2.0b2
2.1b1
```

Release candidates are identified by appending the letters RC to the major and minor release numbers followed by the build number:

```
MAJOR.MINORrcBUILD
2.0rc1
2.0rc2
2.1rc1
```

Subversion copies are created for release using the version number in the *releases* directory:

```
/public/fltk/fltk/releases/release-2.0.0b1
/public/fltk/fltk/releases/release-2.0.0rc1
/public/fltk/fltk/releases/release-2.0.0
```

Subversion copies are created for every complete major or minor release in the *branches* directory:

```
/public/fltk/fltk/branches/branch-2.0
/public/fltk/fltk/branches/branch-2.1
```

New development then continues in the *trunk* directory with fixes backported to the corresponding *branches* directory as needed.

Each change that corrects a fault in a software sub-system increments the patch release number. If a change affects the overall software design of FLTK then the minor release number will be incremented and the patch release number reset to 0. If FLTK is completely redesigned the major release number will be incremented and the minor and patch release numbers reset to 0:

```
2.0b1    First beta of 2.0
2.0rc1   First release candidate of 2.0
2.0rc2   Second release candidate of 2.0
```

Configuration Management Plan - Fast Light Toolkit (FLTK)

2.0.0	Production release of 2.0
2.0.1	First patch release of 2.0
2.0.2	Second patch release of 2.0
2.1b1	First beta of 2.1
2.1rc1	First release candidate of 2.1
2.1rc2	Second release candidate of 2.1
2.1.0	Production release of 2.1
3.0b1	First beta of 3.0
3.0rc1	First release candidate of 3.0
3.0rc2	Second release candidate of 3.0
3.0.0	Production release of 3.0
3.0.1	First patch release of 3.0

Generation

Software releases shall be generated for each successfully completed software trouble report. All object and executable files shall be deleted prior to performing a full build to ensure that source files are recompiled.

Testing

Software testing shall be conducted according to the FLTK Software Test Plan (*Editor's note: to be written, along with an automated/semi-automated test framework*). Failed tests cause STRs to be generated to correct the problems found.

Releases

When testing has been completed successfully a new distribution image is created by copying the current *trunk* or *branches* directory to the *releases* directory as specified previously. **No release shall contain software that has not passed the appropriate software tests.** Four types of releases are used, beta, release candidate, production, and patch, and are released using the following basic schedule:

Week	Version	Description
T-6 weeks	2.0b1	First beta
T-5 weeks	2.0b2	Second beta
T-4 weeks	2.0b3	Third beta
T-3 weeks	2.0rc1	First release candidate
T-2 weeks	2.0rc2	Second release candidate
T-0 weeks	2.0.0	Production
T+N weeks	2.0.1	First patch
T+N weeks	2.0.2	Second patch

Beta releases are typically used prior to new major and minor version releases. At least one release candidate is generated prior to each production release.

Beta Releases

Beta releases are generated when substantial changes have been made that may affect the reliability of the software. Beta releases may cause loss of data, functionality, or services and are provided for testing by qualified individuals.

Configuration Management Plan - Fast Light Toolkit (FLTK)

Beta releases are an OPTIONAL part of the release process and are generated as deemed appropriate by the release coordinator. Functional changes may be included in subsequent beta releases until the first release candidate.

Release Candidates

Release candidates are generated at least two weeks prior to a production release. Release candidates are targeted for end-users that wish to test new functionality or bug fixes prior to the production release. While release candidates are intended to be substantially bug-free, they may still contain defects and/or not compile on specific platforms.

At least one release candidate is REQUIRED prior to any production release. The distribution of a release candidate marks the end of any functional improvements. Release candidates are generated at weekly intervals until all level 4/5 trouble reports are resolved.

Production Releases

Production releases are generated after a successful release candidate and represent a stable release of the software suitable for all users.

Major Releases (Version #.x.x)

Major Releases occur when there's a major rewrite of the code, or a significant redefinition of the API.

Minor Releases (Version x.#.x)

Minor Releases add ABI breaking features/fixes. Rarely breaks the API in any significant way.

Any FLTK_ABI_VERSION code should be resolved in Minor Releases. Any code that looks like:

```
#if FLTK_ABI_VERSION >= 10305
... new ABI breaking code ...
#else
... old non-ABI breaking code ...
#endif
```

..is 'resolved' by *removing* the `#if/#else/#endif` and 'old non-ABI breaking code', so that only the 'new ABI breaking code' is left in its place for default builds.

Patch Releases (Version x.x.#)

Patch releases are generated to fix priority 2-5 STRs. Patch releases may not add additional functionality from priority 1 STRs.

Patch Releases fix small problems that don't break the ABI, and must be API backwards compatible.

ABI breaking fixes/features can be added using FLTK_ABI_VERSION to `#ifdef` out the code from default builds, but can be optionally enabled by end users who need it for testing or for static builds. This can be done by defining this variable to the right ABI version number in `FL/Enumerations.H`, and re-building FLTK and

their apps.

File Management

This section describes how project files and directories are named and managed.

Configuration Management

Source files shall be placed under the control of the Subversion ("SVN") software. Source files shall be "checked in" with each change so that modifications can be tracked, and each check in must reference the applicable STRs that are affected, if any. The following format **must** be used for commit log messages:

```
Summary of the change ("fix OpenGL double-buffer bug") along
with corresponding STRs ("STR #1, STR #6")

foo.cxx:
- Detailed list of changes, by function
- Include summary of design changes ("added new foo struct")

bar.h:
- More detailed changes
```

Documentation on the Subversion software is [available on-line](#).

Directory Structure

Each source file shall be placed a sub-directory corresponding to the software sub-system it belongs to ("fltk", "OpenGL", etc.) To remain compatible with case-insensitive filesystems, no two directory names shall differ only by the case of the letters in the name.

Source Files

Source files shall be documented and formatted as described in the [Coding Standards](#) section. To remain compatible with case-insensitive filesystems, no two filenames shall differ only by the case of the letters in the name.

C source files shall have an extension of ".c". C++ source files shall have an extension of ".cxx". Header files shall have an extension of ".h" unless used for FLTK 1.x compatibility. FLTK 1.x compatibility headers shall have an extension of ".H".

Why use the ".cxx" extension?

C++ source files can have any of the following extensions on various platforms: ".C", ".cc", ".cpp", ".cxx". Only the ".cxx" extension is universally recognized by C++ compilers as a C++ source file - ".C" is not usable on MacOS X and Windows, ".cc" is not usable on Windows, and ".cpp" is historically considered C preprocessor output on UNIX.

Since not all make programs handle C++ source files with the ".cxx" extension, the FLTK build system explicitly defines makefile rules for compiling C++ source files with an extension of ".cxx".

IDE/compiler support source files (project files, workspaces, makefiles, etc.) shall have extensions as required by the IDE/compiler tool.

Header Files

In addition to the source file requirements, all header files must utilize so-called "guard" definitions to prevent multiple inclusion. The guard definitions are named using the full path in the FLTK source tree, e.g.:

- *fltk/fltk.h* becomes `_fltk_fltk_h_`
- *fluid/foo.h* becomes `_fluid_foo_h_`
- *FL/Fl.H* becomes `_FL_Fl_H_`

Any non-alphanumeric (letters and numbers) characters are replaced with the underscore (`_`) character, and leading and trailing underscores are added to limit global namespace pollution.

Makefiles

Makefiles shall be documented and formatted as described in the [Makefile Standards](#) section.

Static makefiles are named "Makefile". Makefiles created by the `autoconf` software are named "Makefile.in". The common include file for all makefiles is named "makeinclude.in".

Coding Standards

The following is a guide to the coding style that must be used when adding or modifying code in FLTK. Most of this should be obvious from looking at the code, but here it all is in one spot.

General Coding Style

The FLTK code basically follows the K&R coding style. While many of the developers are not entirely satisfied with this coding style, no one has volunteered to change all of the FLTK source code (currently about 54,000 lines of code!) to a new style.

The K&R coding style can be summarized with the following example code:

```
int
function(int arg) {
    if (arg != 10) {
        printf("arg = %d\n", arg);
        return (0);
    } else {
        return 1;
    }
}

int function2(int arg) {
```

Configuration Management Plan - Fast Light Toolkit (FLTK)

```
for (int i=0; i<arg; i++) {
    stuff();
}
while (something) {
    stuff();
}
switch (arg) {
    case 0:
        stuff_here();
        break;
    case 1: {
        int var;
        stuff_here();
        break;
    }
    case 2:
        stuff();
        /* FALLTHROUGH */
    case 3: simple_stuff1(); break;
    case 4: simple_stuff2(); break;
    default:
        break;
}
return (0);
}
```

All curly braces must open on the same line as the enclosing statement, and close at the same level of indentation. Each block of code must be indented 2 spaces. **If you use tabs, they must be set at every 8 character columns**; this is different than the Microsoft standard of 4, but you can change that to match UNIX. A space also follows all reserved words.

Source File Documentation

Each source file must start with the standard FLTK header containing the Subversion "\$Id\$" keyword, description of the file, and FLTK copyright and license notice:

```
//
// "$Id$"
//
// some descriptive text for the Fast Light Tool Kit (FLTK).
//
// Copyright 1998-2005 by Bill Spitzak and others.
//
// This library is free software; you can redistribute it and/or
// modify it under the terms of the GNU Library General Public
// License as published by the Free Software Foundation; either
// version 2 of the License, or (at your option) any later version.
//
// This library is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
// Library General Public License for more details.
//
// You should have received a copy of the GNU Library General Public
// License along with this library; if not, write to the Free Software
// Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307
// USA.
//
// Please report all bugs and problems to:
```

Configuration Management Plan - Fast Light Toolkit (FLTK)

```
//  
//      http://www.fltk.org/str.php  
//
```

or the equivalent comment blocking using the C comment delimiters. The end of each source file must have a comment saying:

```
//  
// End of "$Id$".  
//
```

or:

```
/*  
 * End of "$Id$".  
*/
```

The purpose of the trailer is to indicate the end of the source file so that truncations are immediately obvious.

Documentation (Comments) *

FLTK 2.0 and FLTK 1.3 and up uses Doxygen with the JavaDoc comment style to document all classes, structures, enumerations, methods, and functions. Doxygen comments are **mandatory** for all FLTK header and source files, and no FLTK release will be made without complete documentation of public APIs. Here is an example of the Doxygen comment style:

```
/**  
    The Foo class implements the Foo widget for FLTK.  
  
    This description text appears in the documentation for  
    the class and may include HTML tags as desired.  
*/  
  
class FL_EXPORT Foo : public Widget {  
    int private_data_;  
  
public:  
    /**  
        Creates a Foo widget with the given position and label.  
  
        This description text appears in the documentation for the  
        method's implementation.  
  
        References to parameters \p X, \p Y, \p W, \p H are  
        mentioned this way.  
  
        \param[in] X,Y,W,H Position and size of widget  
        \param[in] L Optional label (default is 0 for no label)  
    */  
    Foo(int X, int Y, int W, int H, const char *L = 0) {  
        ..implementation here..  
    }  
};
```

Essentially, a comment starting with `/**` before the class or method defines the documentation for that class or method. These comments should appear in the header file for classes and inline methods and in the code file for non-inline methods.

Configuration Management Plan - Fast Light Toolkit (FLTK)

In addition to Doxygen comments, block comments must be used liberally in the code to describe what is being done. If what you are doing is not "intuitively obvious to a casual observer", add a comment! Remember, you're not the only one that has to read, maintain, and debug the code.

Never use C++ comments in C code files or in header files that may be included from a C program. (Otherwise builds on strict platforms like SGI will fail). Normally, fltk C files have ".c" and ".h" file extensions, and C++ have ".cxx" and ".H". Currently there are a few exceptions; filename.H and Fl_Exports.H both get interpreted by C and C++, so you *must* use C style comments in those.

General Developer Recommendations *

Most important rule: **Put Doxygen comments where the code's implementation is.** This means don't put the docs with the prototypes in the .H file, unless that's where the code is implemented.

- class, typedef, enum, and inline docs go in the headers
- Most other docs go in the source files
- For doxygen syntax in C++ files, use:

```
/** for standard doxygen comments */
///< for short single line post-declaration doxygen comments
```

- For doxygen syntax in C files, use:

```
/** for standard doxygen comments */
/**< for short single line post-declaration doxygen comments */
```

- Use \p for parameters citation in the description
- Use \param[in] xxx and \param[out] xxx for input/output parameters.
- Don't use doxygen tags between the \htmlonly and \endhtmlonly pair of tags.
- When commenting out code or writing non-doxygen comments, be sure not to accidentally use doxygen comment styles, or your comments will be published..! Beware doxygen recognizes other comment styles for itself:

```
/*! beware */
/*@ beware */
//! beware
//@ beware
```

There may be others. For this reason, *follow all non-doxygen comment leaders with a space* to avoid accidental doxygen parsing:

```
/* safe from doxygen */
// safe from doxygen
^
/|\
|
Space immediately after comment characters
```

- Note: Several characters are 'special' within doxygen comments, and must be escaped with a backslash to appear in the docs correctly. Some of these are:

```
\<    -- disambiguates html tags
\>    -- " "
\&    -- " "
```

Configuration Management Plan - Fast Light Toolkit (FLTK)

```
\@    -- disambiguates JavaDoc doxygen comments
\$$   -- disambiguates environment variable expansions
\#    -- disambiguates references to documented entities
\%    -- prevents auto-linking
\\    -- escapes the escape character
```

- Itemized lists can be specified two ways; both work, left is preferred:

- Preferred -	- Old -
<pre>/** Here's a bullet list: - Apples - Oranges Here's a numbered list: -# First thing -# Second thing */</pre>	<pre>/** Here's a bullet list: Apples Oranges Here's a numbered list: First thing Second thing */</pre>

Documenting Temporary Code or Issues

Temporary code and code that has a known issue **MUST** be documented in-line with the following (Doxygen) comment style:

```
/** \todo this code is temporary */
```

`\todo` items are listed by Doxygen making it easy to locate any code that has an outstanding issue or code that should be removed or commented out prior to a release.

Documenting Classes and Structures

Classes and structures start with a comment block that looks like the following:

```
/**
    A brief description of the class/structure.

    A complete description of the class/structure.
*/
class MyClass {
};
```

Documenting Enumerations

Enumerations start with a comment block that looks like the following:

```
/**
    A brief description of the enumeration.

    A complete description of the enumeration.
*/
enum MyEnum {
    ...
};
```

Configuration Management Plan - Fast Light Toolkit (FLTK)

Each enumeration value must be documented in-line next to the corresponding definition as follows:

```
/* C++ STYLE */
enum MyEnum {
    BLACK,    ///< The color black.
    RED,      ///< The color red.
    GREEN,    ///< The color green.
    YELLOW,   ///< The color yellow.
    BLUE,     ///< The color blue.
};
```

If the enum is included in a C file, be sure to use C style commenting:

```
/* C STYLE */
enum MyEnum {
    BLACK,    /**< The color black. */
    RED,      /**< The color red. */
    GREEN,    /**< The color green. */
    YELLOW,   /**< The color yellow. */
    BLUE,     /**< The color blue. */
};
```

Documenting Functions and Methods

Functions and methods start with a comment block that looks like the following:

```
/**
    A brief description of the function/method.

    A complete description of the function/method.
    Optional passing mention of parameter \p a and \p out1.

    Optional code example goes here if needed:
    \code
    ..code showing how to use it..
    \endcode

    \param[in] a Description of input variable a
    \param[in] x,y Description of input variables x and y in one comment
    \param[out] out1 Description of output variable out1
    \param[out] out2 Description of output variable out2
    \return 0 on success, -1 on error
    \see other_func1(), other_func2()
*/
int my_function(int a, int x, int y, float &out1, float &out2) {
    ...implementation...
}
```

Some details on the above:

Parameters

Use **\param** to document function/method parameters using either of the following formats, the latter being preferred:

```
\param var Some description
\param[in|out] var Some description.
```


Configuration Management Plan - Fast Light Toolkit (FLTK)

Mention of parameters in docs should use "`\p varname`". (Use `\p` instead of `\a`)

Note: Doxygen checks your `\param` variable names against the actual function signatures in your code. It does NOT check `\p` names for consistency.

Return Values

Use `\return` to document return values. Omit this if there is no return value. **Reference related methods**

Use `\see` to help the reader find related methods. (Methods are sorted alphabetically by doxygen, so 'related' methods might not appear together)

Locate `\see` references below `\param[]` and `\return` as shown in the above example.

Code examples

Use `\code` and `\endcode` when code examples are needed. Text within will be exempt from html and doxygen escape code parsing, so you don't have to escape characters `<`, `>`, `&`, etc. as you would normally.

Be careful not to embed C style comments within `\code` or it will break the outer doxygen comment block. (A good reason to always test build the code base before committing documentation-only mods)

Where to put docs

Function/method documentation must be placed next to the corresponding code. (Rule: "Put the docs where the code implementation is.") Comments for in-line functions and methods are placed in the header file where they're defined.

Documenting Members and Variables

Members and variables can be documented in one of two ways; in block comment form:

```
/**
 * A brief doxygen description of the member/variable.
 *
 * A complete description of the member/variable.
 * More text goes here..
 */
int my_variable_;
```

or in the preferred form as in-line comments as follows:

```
int my_variable1_;    ///< C++ file's brief doxygen description of the member/variable
int my_variable2_;    ///< C file's brief doxygen description of the member/variable */
```

Methodology, Algorithms, Etc.

The goal of FLTK is to provide a robust GUI toolkit that is small, fast, and reliable. **All** public API functions and methods must be documented with the valid values for all input parameters - NULL pointers, number ranges, etc. - and no public API function may have undefined behaviors. Input validation should be performed only when the function or method is able to return an error to the caller.

Configuration Management Plan - Fast Light Toolkit (FLTK)

When solving a particular problem, whether you are writing a widget or adding functionality to the library, please consider the following guidelines:

1. Choose the small, simple, easy-to-test algorithm over a more complex, larger one that is harder to debug and maintain.
2. Choose the fastest algorithm that satisfies #1
3. Break complex solutions into smaller, more manageable pieces.
4. If functionality can be separated from the main part of the FLTK library, separate it. The idea is to keep the FLTK "core" as small as possible so that programs use memory proportionate to their complexity rather than starting big.
5. Choose a general-purpose solution over a single-purpose solution, i.e. don't limit your design to what *you* think something will be used for.
6. Don't rely on functionality available on a particular platform or compiler; this ties in with #5.

C++ Portability

Since FLTK is targeted at platforms which often lack complete ISO C++ support or have limited memory, all C++ code in FLTK must use a subset of ISO C++. FLTK These restrictions shall be reviewed prior to each major release of FLTK.

FLTK 1.1.x Restrictions

The following C++ features may be not used in FLTK 1.1.x code:

- Exceptions
- Namespaces
- Standard C++ headers and library
- Templates
- `static_cast`, `dynamic_cast`, `const_cast`

FLTK 2.0.x Restrictions

The following C++ features may be not used in FLTK 2.0.x code:

- Exceptions
- Standard C++ headers and library
- Templates
- `dynamic_cast`

The `static_cast` and `const_cast` keywords must be used in 2.0.x code when casting pointers of different types. The `dynamic_cast` keyword must not be used since run-time typing features are not be available at all times.

Source File Naming

The current practice is to use an extension of ".c" for C source files, ".h" for C header files, ".cxx" for C++ source files, and ".H" for C++ header files in the "FL" directory (".h" otherwise.)

Function/Method/Variable Naming

All public (exported) functions and variables must be placed in the "fltk" namespace. Except for constructor and destructor methods, the names consist of lowercase words separated by the underscore ("_"), e.g. "fltk::some_variable" and "text_color()". Private member variables of classes end with an extra underscore, e.g. "text_size_".

Structure/Class Naming

All public (exported) structures and classes must be placed in the "fltk" namespace and consist of capitalized words without the underscore, e.g. "fltk::SuperWidget".

Private members of classes must end with a trailing underscore ("_") and have corresponding public access methods without the underscore as applicable, e.g. "text_size_" and "text_size()".

Constant/Enumeration Naming

Public enumerations and constant variables must be placed inside the "fltk" namespace and consist of UPPERCASE WORDS separated by the underscore ("_"), e.g. "ALIGN_LEFT", "COLOR_RED", etc. Enumeration type names consist of capitalized words without underscores, e.g. "MyEnum". #define constants are prohibited aside from the include guard definitions.

Preprocessor Variables

File config.h and the C++ compilers define a few preprocessor variables that help organizing platform-specific code and control access to a few internal classes. Only code internal to the FLTK library can include the config.h header file. Thus, FLTK header files that are part of the public API must not, directly or indirectly, include config.h.

- WIN32 identifies the MS-Windows platform (both for the 32- and 64-bit versions). Don't use _WIN32.
- __CYGWIN__ is defined when FLTK runs on the MS-Windows OS but uses Cygwin's POSIX emulation features (cygwin1.dll). [more to come...]
- __APPLE__ identifies the Mac OS X platform.
- __APPLE_QUARTZ__ is defined by config.h for the Mac OS X platform. At present, use of __APPLE_QUARTZ__ is equivalent to using __APPLE__. This may change in the future if other graphics systems than Quartz are supported on the Mac OS platform.
- USE_X11 is defined by config.h when Xlib is the graphics system used. Thus, USE_X11 is defined on all Unix and Linux platforms, and on Windows, if configure used --enable-cygwin **and** --enable-x11. Xlib-specific code is also often delimited without reference to the USE_X11 variable (thus without the requirement to include config.h) as follows:

```
#if defined(WIN32)
#elif defined(__APPLE__)
#else
.. Xlib specific code ...
#endif
```

- USE_XFT is defined by config.h when USE_X11 is defined. It is set to 1 when the Xft library of scalable, anti-aliased fonts is used, and to 0 otherwise.

Configuration Management Plan - Fast Light Toolkit (FLTK)

- `FL_LIBRARY` is defined by all FLTK library build tools when the FLTK library itself is compiled. Application program developers should not define it when compiling their programs.
- `FL_INTERNALS`. Application program developers can define this variable to get access to some internal classes (e.g., the `Fl_X` class) if they need it. APIs to these internal classes are highly subject to changes, though.
- `FL_DOXYGEN` is defined when the Doxygen program that builds the FLTK documentation processes the source code. This variable has two major uses.
 1. `#ifndef FL_DOXYGEN / #endif` allows to hide code from Doxygen.
 2. The Doxygen program does not define the platform-specific variables `__APPLE__` or `WIN32` (even if it's run on Mac OS or MSWindows). Thus, platform-specific (say, Mac-specific) code must be bracketted as follows to be seen by Doxygen :

```
#if defined(__APPLE__) || defined(FL_DOXYGEN)
... Doxygen-visible, Mac-specific code ...
#endif
```

- `FLTK_ABI_VERSION` is used to allow developers to implement ABI breaking code in Patch Releases. Normally unset, can be set by users or devs in `FL/Enumerations.H` to enable ABI breaking features for testing or use by end users in static builds of FLTK.

When set, the variable's value is expected to be the integer representation of the FLTK version number, where the Minor and Patch numbers are padded to 2 digits to allow for numbers 1 thru 99, e.g.

```
#define FLTK_ABI_VERSION 10302    // FLTK version 1.3.2
..'1' is the major version (no padding; avoids octal issues)
..'03' is the minor version (2 digit padding)
..'02' is the patch version (2 digit padding)
```

ABI breaking features are by default `#ifdef`'ed out with this variable during patch releases, and are merged in by developers during the next Minor Release.

Example: If the current patch release is 1.3.1, and the developer adds an ABI-breaking fix to what will be the next 1.3.2 release, then the new code would be implemented as:

```
#if FLTK_ABI_VERSION >= 10302    // FLTK 1.3.2, the next patch release #
... new ABI breaking code ...
#else
... old non-ABI breaking (default builds) ...
#endif
```

This variable solves several issues:

- Allows ABI breaking code to be implemented at any time by developers.
- Gets fixes into SVN sooner, so users can see, test and access it.
- Testing ABI features during Patch Releases increases the stability of Minor Releases.
- Prevents devs having to defer ABI breaking code to the small window of time preceding Minor Releases.

Miscellaneous

When using `switch - case` statements, and your `case` statement does not end in `break` in order to fall through to next `case` statement, the comment `/* FALLTHROUGH */` should be added where the `break` statement would be.

Makefile Standards

The following is a guide to the makefile-based build system used by FLTK. These standards have been developed over the years to allow FLTK to be built on as many systems and environments as possible.

General Organization

The FLTK source code is organized functionally into a top-level makefile, include file, and subdirectories each with their own makefile and dependencies files:

```

Makefile.in
config.in
configure.in
makeinclude.in

FL
    Makefile.in
    *.H

fltk
    *.h

fluid
    Makefile
    makedepend
    *.h
    *.c
    *.cxx

src
    Makefile
    makedepend
    *.h
    *.c
    *.cxx

test
    Makefile
    makedepend
    *.h
    *.c
    *.cxx

```

The ".in" files are template files for the `autoconf` software and are used to generate a static version of the corresponding file.

Makefile Documentation

Each make file must start with the standard FLTK header containing the Subversion "\$Id\$" keyword, description of the file, and FLTK copyright and license notice:

```

#
# "$Id$"
#
# some descriptive text for the Fast Light Tool Kit (FLTK).
#

```

Configuration Management Plan - Fast Light Toolkit (FLTK)

```
# Copyright 1998-2005 by Bill Spitzak and others.
#
# This library is free software; you can redistribute it and/or
# modify it under the terms of the GNU Library General Public
# License as published by the Free Software Foundation; either
# version 2 of the License, or (at your option) any later version.
#
# This library is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
# Library General Public License for more details.
#
# You should have received a copy of the GNU Library General Public
# License along with this library; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307
# USA.
#
# Please report all bugs and problems to:
#
#     http://www.fltk.org/str.php
#
```

The end of each makefile must have a comment saying:

```
#
# End of "$Id$".
#
```

The purpose of the trailer is to indicate the end of the makefile so that truncations are immediately obvious.

Portable Makefile Construction

FLTK uses a common subset of make program syntax to ensure that the software can be compiled "out of the box" on as many systems as possible. The following is a list of assumptions we follow when constructing makefiles:

- **Targets;** we assume that the make program supports the notion of simple targets of the form "name:" that perform tab-indented commands that follow the target, e.g.:

```
target:
TAB▶ target commands
```

- **Dependencies;** we assume that the make program supports recursive dependencies on targets, e.g.:

```
target: foo bar
TAB▶ target commands
```

```
foo: bla
TAB▶ foo commands
```

```
bar:
TAB▶ bar commands
```

```
bla:
TAB▶ bla commands
```

- **Variable Definition;** we assume that the make program supports variable definition on the command-line or in the makefile using the following form:

Configuration Management Plan - Fast Light Toolkit (FLTK)

name=value

- **Variable Substitution**; we assume that the make program supports variable substitution using the following forms:
 - ◆ `$(name)`; substitutes the value of "name",
 - ◆ `$(name:.old=.new)`; substitutes the value of "name" with the filename extensions ".old" changed to ".new",
 - ◆ `$(MAKEFLAGS)`; substitutes the command-line options passed to the program without the leading hyphen (-),
 - ◆ `$$`; substitutes a single `$` character,
 - ◆ `$<`; substitutes the current source file or dependency, and
 - ◆ `$@`; substitutes the current target name.
- **Suffixes**; we assume that the make program supports filename suffixes with assumed dependencies, e.g.:

```
.SUFFIXES: .c .o
.c.o:
TAB▶ $ (CC) $(CFLAGS) -o $@ -c $<
```

- **Include Files**; we assume that the make program supports the `include` directive, e.g.:

```
include ../makeinclude
include makedepend
```

- **Comments**; we assume that comments begin with a `#` character and proceed to the end of the current line.
- **Line Length**; we assume that there is no practical limit to the length of lines.
- **Continuation of long lines**; we assume that the `\` character may be placed at the end of a line to concatenate two or more lines in a makefile to form a single long line.
- **Shell**; we assume a POSIX-compatible shell is present on the build system.

Standard Variables

The following variables are defined in the "makeinclude" file generated by the `autoconf` software:

- `AR`; the library archiver command,
- `ARFLAGS`; options for the library archiver command,
- `BUILDROOT`; optional installation prefix,
- `CAT1EXT`; extension for formatted man pages in section 1,
- `CAT3EXT`; extension for formatted man pages in section 3,
- `CC`; the C compiler command,
- `CFLAGS`; options for the C compiler command,
- `CXX`; the C++ compiler command,
- `CXXFLAGS`; options for the C++ compiler command,
- `DOXYGEN`; the `doxygen` command,
- `DSOCOMMAND`; the shared library building command,
- `EXEEXT`; the extension for executable programs,
- `FLUID`; the `FLUID` executable to install,
- `GLDEMOS`; the OpenGL demos to build,
- `GLLIBS`; libraries for OpenGL programs,
- `HTMLDOC`; the `htmldoc` command,
- `IMAGEDIRS`; image library directories to build,
- `IMGLIBS`; libraries for image programs,
- `INSTALL`; the `install` command,

Configuration Management Plan - Fast Light Toolkit (FLTK)

- `INSTALL_BIN`; the program installation command,
- `INSTALL_DATA`; the data file installation command,
- `INSTALL_DIR`; the directory installation command,
- `INSTALL_LIB`; the library installation command,
- `INSTALL_MAN`; the documentation installation command,
- `INSTALL_SCRIPT`; the shell script installation command,
- `LDFLAGS`; options for the linker,
- `LIBNAME`; the name of the FLTK library to install,
- `LIBS`; libraries for all programs,
- `LN`; the `ln` command,
- `MAKEDEPEND`; the `makedepend` command,
- `MKDIR`; the `mkdir` command,
- `NROFF`; the `nroff` command,
- `OPTIM`; common compiler optimization options,
- `POSTBUILD`; the post build command to run (MacOS X),
- `RM`; the `rm` command,
- `RMDIR`; the `rmdir` command,
- `SHAREDLIBS`; shared libraries for installed programs,
- `SHELL`; the `sh` (POSIX shell) command,
- `STRIP`; the `strip` command,
- `THREADS`; the threading demos to build,
- `bindir`; the binary installation directory,
- `datadir`; the data file installation directory,
- `exec_prefix`; the installation prefix for executable files,
- `libdir`; the library installation directory,
- `mandir`; the man page installation directory,
- `prefix`; the installation prefix for non-executable files, and
- `srcdir`; the source directory.

Standard Targets

The following standard targets must be defined in each makefile:

- `all`; creates all target programs, libraries, and documentation files,
- `clean`; removes all target programs, libraries, documentation files, and object files,
- `depend`; generates automatic dependencies for any C or C++ source files (also see "[Dependencies](#)"),
- `distclean`; removes autoconf-generated files in addition to those removed by the "clean" target,
- `install`; installs all distribution files in their corresponding locations (also see "[Install/Uninstall Support](#)"),
- `uninstall`; removes all distribution files from their corresponding locations (also see "[Install/Uninstall Support](#)"), and
- `unittest`; runs any unit tests that have been created for the corresponding code and programs.

Object Files

Object files (the result of compiling a C or C++ source file) have the extension ".o".

Programs

Program files (the result of linking object files and libraries together to form an executable file) have the extension specified by the `$(EXEEXT)` variable. A typical program target looks like:

```
program$(EXEEXT): $(OBJECTS)
TAB▶ echo Linking $@...
TAB▶ $(CXX) $(LDFLAGS) -o $@ $(OBJECTS) $(LIBS)
```

Static Libraries

Static libraries have a prefix of "lib" and the extension ".a". A typical static library target looks like:

```
libname.a: $(OBJECTS)
TAB▶ echo Creating $@...
TAB▶ $(RM) $@
TAB▶ $(AR) $(ARFLAGS) $@ $(OBJECTS)
TAB▶ $(RANLIB) $@
```

Shared Libraries

Shared libraries have a prefix of "lib" and the extension ".dylib", ".sl", ".so", or "_s.a" depending on the operating system. A typical shared library is composed of several targets that look like:

```
libname.so: $(OBJECTS)
TAB▶ echo $(DSOCOMMAND) libname.so.$(DSOVERSION) ...
TAB▶ $(DSOCOMMAND) libname.so.$(DSOVERSION) $(OBJECTS)
TAB▶ $(RM) libname.so libname.so.$(DSOMAJOR)
TAB▶ $(LN) libname.so.$(DSOVERSION) libname.so.$(DSOMAJOR)
TAB▶ $(LN) libname.so.$(DSOVERSION) libname.so

libname.sl: $(OBJECTS)
TAB▶ echo $(DSOCOMMAND) libname.sl.$(DSOVERSION) ...
TAB▶ $(DSOCOMMAND) libname.sl.$(DSOVERSION) $(OBJECTS)
TAB▶ $(RM) libname.sl libname.sl.$(DSOMAJOR)
TAB▶ $(LN) libname.sl.$(DSOVERSION) libname.sl.$(DSOMAJOR)
TAB▶ $(LN) libname.sl.$(DSOVERSION) libname.sl

libname.dylib: $(OBJECTS)
TAB▶ echo $(DSOCOMMAND) libname.$(DSOVERSION).dylib ...
TAB▶ $(DSOCOMMAND) libname.$(DSOVERSION).dylib \
TAB▶ TAB▶ -install_name $(libdir)/libname.$(DSOMAJOR).dylib \
TAB▶ TAB▶ -current_version libname.$(DSOVERSION).dylib \
TAB▶ TAB▶ -compatibility_version $(DSOMAJOR).0 \
TAB▶ TAB▶ $(OBJECTS) $(LIBS)
TAB▶ $(RM) libname.dylib
TAB▶ $(RM) libname.$(DSOMAJOR).dylib
TAB▶ $(LN) libname.$(DSOVERSION).dylib libname.$(DSOMAJOR).dylib
TAB▶ $(LN) libname.$(DSOVERSION).dylib libname.dylib

libname_s.a: $(OBJECTS)
TAB▶ echo $(DSOCOMMAND) libname_s.o ...
TAB▶ $(DSOCOMMAND) libname_s.o $(OBJECTS) $(LIBS)
TAB▶ echo $(LIBCOMMAND) libname_s.a libname_s.o
TAB▶ $(RM) $@
TAB▶ $(LIBCOMMAND) libname_s.a libname_s.o
TAB▶ $(CHMOD) +x libname_s.a
```

Dependencies

Static dependencies are expressed in each makefile following the target, for example:

```
foo: bar
```

Static dependencies shall only be used when it is not possible to automatically generate them. Automatic dependencies are stored in a file named "makedepend" and included at the end of the makefile. The following "depend" target rule shall be used to create the automatic dependencies:

```
depend: $(CPPFILES) $(CFILES)
TAB▶ $ (MAKEDEPEND) -Y -I.. -f makedepend $(CPPFILES) $(CFILES)
```

We only regenerate the automatic dependencies on a Linux system and express any non-Linux dependencies statically in the makefile.

Install/Uninstall Support

All makefiles must contain install and uninstall rules which install or remove the corresponding software. These rules must use the `$(BUILDROOT)` variable as a prefix to any installation directory so that FLTK can be installed in a temporary location for packaging by programs like `rpmbuild`.

The `$(INSTALL_BIN)`, `$(INSTALL_DATA)`, `$(INSTALL_DIR)`, `$(INSTALL_LIB)`, `$(INSTALL_MAN)`, and `$(INSTALL_SCRIPT)` variables must be used when installing files so that the proper ownership and permissions are set on the installed files.

The `$(RANLIB)` command must be run on any static libraries after installation since the symbol table is invalidated when the library is copied on some platforms.

Developer Reference

Comment Style

Developers for 1.3 settled on the `/** .. */` format for these reasons:

```
erco 3/13/09:
    I do like when (*)s run down left margin of all comments; easier
    to differentiate comments from code in large doc blocks.
matt 3/14/09:
    Yes, same here. I usually align them in the second column.
erco 3/15/09:
    Duncan doesn't like (*)s down the left because it complicates
    paragraph reformatting.. passing that on.
erco 3/15/09:
    Albrecht says this was already discussed and decision was *no stars*
    down the left, so I modified the examples here to follow this rule.
erco 3/15/09:
    Note: fltk2 uses QT /*! style comments, whereas fltk1 uses /**
    as described above. Should standard reflect this?
erco 07/18/10:
    We seem to be going with /** style comments, no (*)s running down
    left margin (as per Duncan's sugg).
```

General Developer Recommendations

Many of the notes in this section are from Fabien's TODO.doc, and seem consistent with the actual fltk docs during the 1.1.x-to-1.3.x doxygenification transition.

**** This space intentionally left blank ****