# Contributing to libui

libui is an open source project that openly accepts contributions. I appreciate your help!

## Rules for contributing code

While libui is open to contributions, a number of recent, significantly large contributions and uncontributed forks have recently surfaced that do not present themselves in a form that makes it easy for libui to accept them. In order to give your contribution a high chance of being accepted into libui, please keep the following in mind as you prepare your contribution.

### Commit messages and pull request description

libui does not enforce rules about the length or detail that a commit message. I'm not looking for an essay. However, single-word descriptions of nontrivial changes are \*not\* acceptable. I should be able to get a glimpse of what a commit does from the commit message, even if it's just one sentence to describe a trivial change. (Yes, I know I haven't followed this rule strictly myself, but I try not to break it too.) And a commit message should encompass everything; typically, I make a number of incremental commits toward a feature, so the commit messages don't have to be too long to explain everything.

Your pull request description, on the other hand, must be a summary of the sum total of all the changes made to libui. Don't just drop a pull request on me with a one-line-long elevator pitch of what you added. Describe your proposed API changes, implementation requirements, and any important consequences of your work.

### Code formatting

libui uses K&R C formatting rules for overall code structure: spaces after keywords like `if`, `{` on the same line as a statement with a space, `{` on its own line after a function or method signature (even those inside the class body), no space after the name of a function, etc.

Use hard tabs, NOT spaces, for indentation. I use a proportional-width font and my text editor doesn't set tabs to a multiple of the space width, so I \*will\* be able to tell. If you use a fixed-width font, I suggest setting a tab width of 4 spaces per tab, but don't put diagrams in comments with hard tabs, because not everyone does this.

Expressions should have a space around binary operators, and use parentheses where it would help humans gather the meaning of an expression, regardless of whether a computer could tell what is correct.

When breaking expressions into multiple lines, always break \*after\* an operator, such as `,` or `&&`.

There should be a newline between a function's variables and a function's code. After that, you can place newlines to delimit different parts of a function, but don't go crazy.

In the event you are unsure of something, refer to existing libui code for examples. I may wind up fixing minor details later anyway, so don't fret about getting minor details right the first time.

### Naming

libui uses camel-case for naming, with a handful of very specific exceptions (namely GObject method names, where GObject itself enforces the naming convention).

All public API names should begin with `ui` and followed by a capital letter. All public struct field names should begin with a capital letter. This is identical to the visibiilty rules of Go, assuming a package name of `ui`.

Private API names — specifcally those used by more than one source file — should begin with `uipriv` and be followed by a capital letter. This avoids namespace collisions in static libraries.

Static functions and static objects do not have naming restrictions.

Acronyms should \*\*NOT\*\* be mixed-case. `http` for the first word in a camel-case name, `HTTP` for all else, but \*\*NEVER\*\* `Http`. This is possibly the only aspect of the controversial nature of code style that I consider indefensibly stupid.

### API documentation

(TODO I am writing an API documentation tool; once that becomes stable enough I can talk about documenting libui properly. You'll see vestiges of it throughout ui.h, though.)

### Other commenting

(TODO write this part)

### Compatibility

libui takes backward compatibility seriously. Your code should not break the current compatibility requirements. All platforms provide a series of macros, defined in the various `uipriv\_\*.h` files (or `winapi.hpp` on Windows), that specify the minimum required version. If you find yourself needing to remove these or ignore resultant warnings or errors, you're probably breaking compatibility.

Choosing to drop older versions of Windows, GTK+, and OS X that I could have easily continued to support was not done lightly. If you want to discuss dropping support for an older version of any of these for the benefit of libui, file an issue pleading your case (see below).

GTK+ versions are harder to drop because I am limited by Linux distribution packaging. In general, I will consider bumping GTK+ versions on a new Ubuntu LTS release, choosing the earliest version available on the major distributions at the time of the \*previous\* Ubuntu LTS release. As of writing, the next milestone will be \*after\* April 2018, and the target GTK+ version appears to be 3.18, judging by Ubuntu 16.04 LTS alone. This may be bumped back depending on other distros (or it may not be bumped at all), but you may wish to keep this in mind as you write.

(TODO talk about future.c/.cpp/.m files)

As for language compatibility, libui is written in C99. I have no intention of changing this.

As for build system compatibility, libui uses CMake 3.1.0. If you wish to bump the version, file an issue pleading your case (but see below).

\*\*If you do plead your case\*\*, keep in mind that "it's old" is not a sufficient reason to drop things. If you can prove that \*\*virtually no one\*\* uses the minimum version anymore, then that is stronger evidence. The best evidence, however, is that not upgrading will hold libui back in some significant way — but beware that there are some things I won't add to libui itself.

### Windows-specific notes

The Windows backend of libui is written in C++ using C++11.

Despite using C++, please refrain from using the following:

- using C++ in ui\_windows.h (this file should still be C compatible)

- smart pointers

- namespaces

- `using namespace`

- ATL, MFC, WTL

The following are not recommended, for consistency with the rest of libui:

- variable declarations anywhere in a function (keep them all at the top)

- `for (int x...` (C++11 foreach syntax is fine, though)

- omitting the `struct` on type names for ordinary structs

The format of a class should be

```c++

class name : public ancestor {

int privateVariable;

// etc.

public:

// public stuff here

};

```

### GTK+-specific notes

Avoid GNU-specific language features. I build with strict C99 conformance.

### OS X-specific notes

Avoid GNU-specific/clang-specific language features. I build with strict C99 conformance.

libui is presently \*\*not\*\* ARC-compliant. Features that require ARC should be avoided for now. I may consider changing this in the future, but it will be a significant change.

To ensure maximum compiler output in the event of a coding error, there should not be any implicit method calls in Objective-C code. For instance, don't do

```objective-c

[[array objectAtIndex:i] method]

```

Instead, cast the result of `objectAtIndex:` to the appropriate type, and then call the method. (TODO learn about, then decide a policy on, soft-generics on things other than `id`)

The format of a class should be

```objective-c

@interface name : parent<protocols> {

// ivars

}

// properties

- (ret)method:(int)arg;

// more methods

@end

@implementation name

- (ret)method:(int)arg

{

// note the lack of semicolon

}

@end

```