# Contributing to wlroots

Contributing just involves sending a pull request. You will probably be more

successful with your contribution if you visit

[#sway-devel](https://webchat.freenode.net/?channels=sway-devel) on

irc.freenode.net upfront and discuss your plans.

Note: rules are made to be broken. Adjust or ignore any/all of these as you see

fit, but be prepared to justify it to your peers.

## Pull Requests

If you already have your own pull request habits, feel free to use them. If you

don't, however, allow me to make a suggestion: feature branches pulled from

upstream. Try this:

1. Fork wlroots

2. `git clone https://github.com/username/wlroots && cd wlroots`

3. `git remote add upstream https://github.com/swaywm/wlroots`

You only need to do this once. You're never going to use your fork's master

branch. Instead, when you start working on a feature, do this:

1. `git fetch upstream`

2. `git checkout -b add-so-and-so-feature upstream/master`

3. Add and commit your changes

4. `git push -u origin add-so-and-so-feature`

5. Make a pull request from your feature branch

When you submit your pull request, your commit log should do most of the talking

when it comes to describing your changes and their motivation. In addition to

this, your pull request's comments will ideally include a test plan that the

reviewers can use to (1) demonstrate the problem on master, if applicable and

(2) verify that the problem no longer exists with your changes applied (or that

your new features work correctly). Document all of the edge cases you're aware

of so we can adequately test them - then verify the test plan yourself before

submitting.

## Commit Messages

Please strive to write good commit messages. Here's some guidelines to follow:

The first line should be limited to 50 characters and should be a sentence that

completes the thought [When applied, this commit will...] \*"Implement

cmd\_move"\* or \*"Fix #742"\* or \*"Improve performance of arrange\_windows on ARM"\*

or similar.

The subsequent lines should be separated from the subject line by a single

blank line, and include optional details. In this you can give justification

for the change, [reference Github

issues](https://help.github.com/articles/closing-issues-via-commit-messages/),

or explain some of the subtler details of your patch. This is important because

when someone finds a line of code they don't understand later, they can use the

`git blame` command to find out what the author was thinking when they wrote

it. It's also easier to review your pull requests if they're separated into

logical commits that have good commit messages and justify themselves in the

extended commit description.

As a good rule of thumb, anything you might put into the pull request

description on Github is probably fair game for going into the extended commit

message as well.

See [here](https://chris.beams.io/posts/git-commit/) for more details.

## Code Review

When your changes are submitted for review, one or more core committers will

look over them. Smaller changes might be merged with little fanfare, but larger

changes will typically see review from several people. Be prepared to receive

some feedback - you may be asked to make changes to your work. Our code review

process is:

1. \*\*Triage\*\* the pull request. Do the commit messages make sense? Is a test

plan necessary and/or present? Add anyone as reviewers that you think should

be there (using the relevant GitHub feature, if you have the permissions, or

with an @mention if necessary).

2. \*\*Review\*\* the code. Look for code style violations, naming convention

violations, buffer overflows, memory leaks, logic errors, non-portable code

(including GNU-isms), etc. For significant changes to the public API, loop in

a couple more people for discussion.

3. \*\*Execute\*\* the test plan, if present.

4. \*\*Merge\*\* the pull request when all reviewers approve.

5. \*\*File\*\* follow-up tickets if appropriate.

## Style Reference

wlroots is written in C with a style similar to the [kernel

style](https://www.kernel.org/doc/Documentation/process/coding-style.rst), but

with a few notable differences.

Try to keep your code conforming to C11 and POSIX as much as possible, and do

not use GNU extensions.

### Brackets

Brackets always go on the same line, including in functions.

Always include brackets for if/while/for, even if it's a single statement.

```c

void function(void) {

if (condition1) {

do\_thing1();

}

if (condition2) {

do\_thing2();

} else {

do\_thing3();

}

}

```

### Indentation

Indentations are a single tab.

For long lines that need to be broken, the continuation line should be indented

with an additional tab.

If the line being broken is opening a new block (functions, if, while, etc.),

the continuation line should be indented with two tabs, so they can't be

misread as being part of the block.

```c

really\_long\_function(argument1, argument2, ...,

argument3, argument4);

if (condition1 && condition2 && ...

condition3 && condition4) {

do\_thing();

}

```

Try to break the line in the place which you think is the most appropriate.

### Line Length

Try to keep your lines under 80 columns, but you can go up to 100 if it

improves readability. Don't break lines indiscriminately, try to find nice

breaking points so your code is easy to read.

### Names

Global function and type names should be prefixed with `wlr\_submodule\_` (e.g.

`struct wlr\_output`, `wlr\_output\_set\_cursor`). For static functions and

types local to a file, the names chosen aren't as important. Local function

names shouldn't have a `wlr\_` prefix.

For include guards, use the header's filename relative to include. Uppercase

all of the characters, and replace any invalid characters with an underscore.

### Construction/Destruction Functions

For functions that are responsible for constructing and destructing an object,

they should be written as a pair of one of two forms:

\* `init`/`finish`: These initialize/deinitialize a type, but are \*\*NOT\*\*

responsible for allocating it. They should accept a pointer to some

pre-allocated memory (e.g. a member of a struct).

\* `create`/`destroy`: These also initialize/deinitialize, but will return a

pointer to a `malloc`ed chunk of memory, and will `free` it in `destroy`.

A destruction function should always be able to accept a NULL pointer or a

zeroed value and exit cleanly; this simplifies error handling a lot.

### Error Codes

For functions not returning a value, they should return a (stdbool.h) bool to

indicated if they succeeded or not.

### Macros

Try to keep the use of macros to a minimum, especially if a function can do the

job. If you do need to use them, try to keep them close to where they're being

used and `#undef` them after.

### Example

```c

struct wlr\_backend \*wlr\_backend\_autocreate(struct wl\_display \*display) {

struct wlr\_backend \*backend;

if (getenv("WAYLAND\_DISPLAY") || getenv("\_WAYLAND\_DISPLAY")) {

backend = attempt\_wl\_backend(display);

if (backend) {

return backend;

}

}

const char \*x11\_display = getenv("DISPLAY");

if (x11\_display) {

return wlr\_x11\_backend\_create(display, x11\_display);

}

// Attempt DRM+libinput

struct wlr\_session \*session = wlr\_session\_create(display);

if (!session) {

wlr\_log(WLR\_ERROR, "Failed to start a DRM session");

return NULL;

}

int gpu = wlr\_session\_find\_gpu(session);

if (gpu == -1) {

wlr\_log(WLR\_ERROR, "Failed to open DRM device");

goto error\_session;

}

backend = wlr\_multi\_backend\_create(session);

if (!backend) {

goto error\_gpu;

}

struct wlr\_backend \*libinput = wlr\_libinput\_backend\_create(display, session);

if (!libinput) {

goto error\_multi;

}

struct wlr\_backend \*drm = wlr\_drm\_backend\_create(display, session, gpu);

if (!drm) {

goto error\_libinput;

}

wlr\_multi\_backend\_add(backend, libinput);

wlr\_multi\_backend\_add(backend, drm);

return backend;

error\_libinput:

wlr\_backend\_destroy(libinput);

error\_multi:

wlr\_backend\_destroy(backend);

error\_gpu:

wlr\_session\_close\_file(session, gpu);

error\_session:

wlr\_session\_destroy(session);

return NULL;

}

```

## Wayland protocol implementation

Each protocol generally lives in a file with the same name, usually containing

at least one struct for each interface in the protocol. For instance,

`xdg\_shell` lives in `types/wlr\_xdg\_shell.h` and has a `wlr\_xdg\_surface` struct.

### Globals

Global interfaces generally have public constructors and destructors. Their

struct has a field holding the `wl\_global` itself, a destroy signal and a

`wl\_display` destroy listener. Example:

```c

struct wlr\_compositor {

struct wl\_global \*global;

struct wl\_listener display\_destroy;

struct {

struct wl\_signal new\_surface;

struct wl\_signal destroy;

} events;

};

```

When the destructor is called, it should emit the destroy signal, remove the

display destroy listener, destroy the `wl\_global` and then destroy the struct.

The destructor can assume all clients and resources have been already

destroyed.

### Resources

Resources are the representation of Wayland objects on the compositor side. They

generally have an associated struct, called the \_object struct\_, stored in their

`user\_data` field.

Object structs can be retrieved from resources via `wl\_resource\_get\_data`. To

prevent bad casts, a safe helper function checking the type of the resource is

used:

```c

static const struct wl\_surface\_interface surface\_impl;

struct wlr\_surface \*wlr\_surface\_from\_resource(struct wl\_resource \*resource) {

assert(wl\_resource\_instance\_of(resource, &wl\_surface\_interface,

&surface\_impl));

return wl\_resource\_get\_user\_data(resource);

}

```

If a pointer to a `wl\_resource` is stored, a resource destroy handler needs to

be registered to clean it up. libwayland will automatically destroy resources

in an arbitrary order when a client is disconnected, the compositor must handle

this correctly.

### Destroying resources

Object structs should only be destroyed when their resource is destroyed, ie.

in the resource destroy handler (set with `wl\_resource\_set\_implementation`).

- If the object has a destructor request: the request handler should just call

`wl\_resource\_destroy` and do nothing else. The compositor must not destroy

resources on its own outside the destructor request handler.

- If the protocol specifies that an object is destroyed when an event is sent:

it's the only case where the compositor is allowed to send the event and then

call `wl\_resource\_destroy`. An example of this is `wl\_callback`.

### Inert resources

Some resources can become inert in situations described in the protocol or when

the compositor decides to get rid of them. All requests made to inert resources

should be ignored, except the destructor. This is achieved by:

1. When the resource becomes inert: destroy the object struct and call

`wl\_resource\_set\_user\_data(resource, NULL)`. Do not destroy the resource.

2. For each request made to a resource that can be inert: add a NULL check to

ignore the request if the resource is inert.

3. When the client calls the destructor request on the resource: call

`wl\_resource\_destroy(resource)` as usual.

4. When the resource is destroyed, if the resource isn't inert, destroy the

object struct.

Example:

```c

// Handles the destroy request

static void subsurface\_handle\_destroy(struct wl\_client \*client,

struct wl\_resource \*resource) {

wl\_resource\_destroy(resource);

}

// Handles a regular request

static void subsurface\_set\_position(struct wl\_client \*client,

struct wl\_resource \*resource, int32\_t x, int32\_t y) {

struct wlr\_subsurface \*subsurface = subsurface\_from\_resource(resource);

if (subsurface == NULL) {

return;

}

}

// Destroys the wlr\_subsurface struct

static void subsurface\_destroy(struct wlr\_subsurface \*subsurface) {

if (subsurface == NULL) {

return;

}

wl\_resource\_set\_user\_data(subsurface->resource, NULL);

free(subsurface);

}

// Resource destroy listener

static void subsurface\_handle\_resource\_destroy(struct wl\_resource \*resource) {

struct wlr\_subsurface \*subsurface = subsurface\_from\_resource(resource);

subsurface\_destroy(subsurface);

}

// Makes the resource inert

static void subsurface\_handle\_surface\_destroy(struct wl\_listener \*listener,

void \*data) {

struct wlr\_subsurface \*subsurface =

wl\_container\_of(listener, subsurface, surface\_destroy);

subsurface\_destroy(subsurface);

}

```