Documentation 1.3

Wayland

The Wayland display server

Kristian Høgsberg

Documentation 1.3 Wayland
The Wayland display server
Edition 1

Author Kristian Høgsberg krh@bitplanet.net

Wayland is a protocol for a compositor to talk to its clients as well as a C library implementation of that protocol. The compositor can be a standalone display server running on Linux kernel modesetting and evdev input devices, an X application, or a Wayland client itself. The clients can be traditional applications, X servers (rootless or fullscreen) or other display servers.

Preface

This document describes the (i) Wayland architecture, (ii) Wayland model of operation and (iii) its library API. Also, the Wayland protocol specification is shown in the Appendix. This document is aimed primarily at Wayland developers and those looking to program with it; it does not cover application development.

There have been many contributors to this document and since this is only the first edition many errors are expected to be found. We appreciate corrections.

Yours,

the Wayland open-source community
November 2012

Acknowledgments

TODO: Kristian has to fill up this with one or two paragraphs and a small "thank you": http://en.wikipedia.org/wiki/Preface

Best,

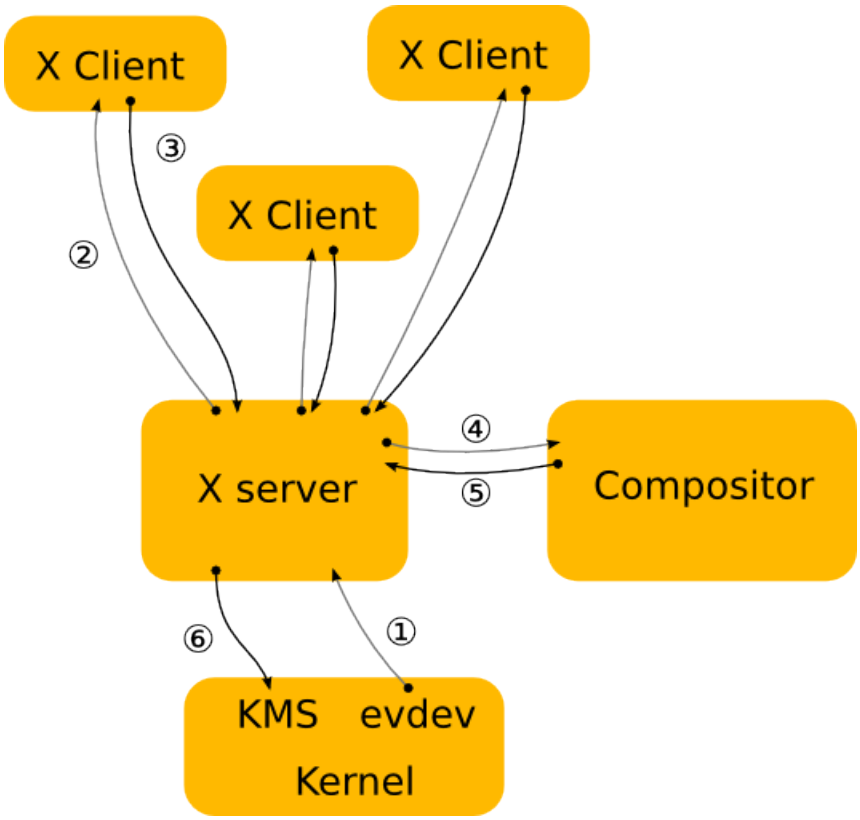
Kristian Høgsberg

Introduction

1.1. Motivation

Most Linux and Unix-based systems rely on the X Window System (or simply *X*) as the low-level protocol for building bitmap graphics interfaces. On these systems, the X stack has grown to encompass functionality arguably belonging in client libraries, helper libraries, or the host operating system kernel. Support for things like PCI resource management, display configuration management, direct rendering, and memory management has been integrated into the X stack, imposing limitations like limited support for standalone applications, duplication in other projects (e.g. the Linux fb layer or the DirectFB project), and high levels of complexity for systems combining multiple elements (for example radeon memory map handling between the fb driver and X driver, or VT switching).

Moreover, X has grown to incorporate modern features like offscreen rendering and scene composition, but subject to the limitations of the X architecture. For example, the X implementation of composition adds additional context switches and makes things like input redirection difficult.



The diagram above illustrates the central role of the X server and compositor in operations, and the steps required to get contents on to the screen.

Over time, X developers came to understand the shortcomings of this approach and worked to split things up. Over the past several years, a lot of functionality has moved out of the X server and into client-side libraries or kernel drivers. One of the first components to move out was font rendering, with freetype and fontconfig providing an alternative to the core X fonts. Direct rendering OpenGL as a graphics driver in a client side library went through some iterations, ending up as DRI2, which abstracted most of the direct rendering buffer management from client code. Then cairo came along and provided a modern 2D rendering library independent of X, and compositing managers took over control of the rendering of the desktop as toolkits like GTK+ and Qt moved away from using X APIs

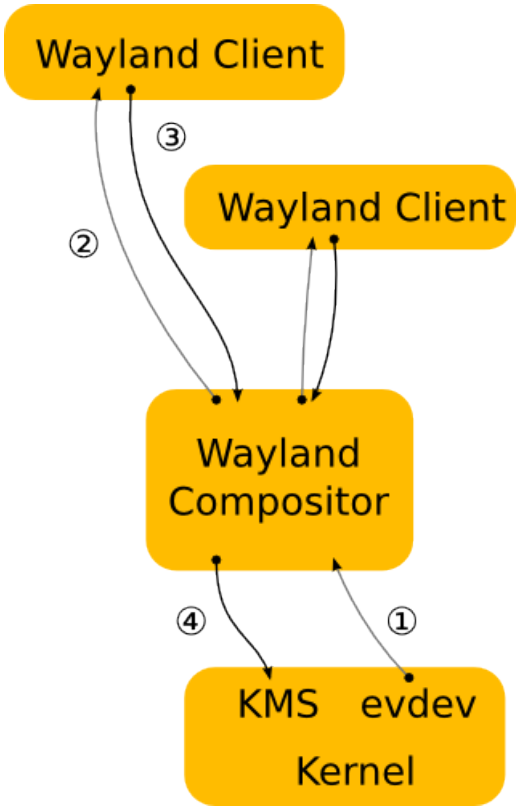
for rendering. Recently, memory and display management have moved to the Linux kernel, further reducing the scope of X and its driver stack. The end result is a highly modular graphics stack.

1.2. The compositing manager as the display server

Wayland is a new display server and compositing protocol, and Weston is the implementation of this protocol which builds on top of all the components above. We are trying to distill out the functionality in the X server that is still used by the modern Linux desktop. This turns out to be not a whole lot. Applications can allocate their own off-screen buffers and render their window contents directly, using hardware accelerated libraries like libGL, or high quality software implementations like those found in Cairo. In the end, what's needed is a way to present the resulting window surface for display, and a way to receive and arbitrate input among multiple clients. This is what Wayland provides, by piecing together the components already in the eco-system in a slightly different way.

X will always be relevant, in the same way Fortran compilers and VRML browsers are, but it's time that we think about moving it out of the critical path and provide it as an optional component for legacy applications.

Overall, the philosophy of Wayland is to provide clients with a way to manage windows and how their contents is displayed. Rendering is left to clients, and system wide memory management interfaces are used to pass buffer handles between clients and the compositing manager.



The figure above illustrates how Wayland clients interact with a Wayland server. Note that window management and composition are handled entirely in the server, significantly reducing complexity while marginally improving performance through reduced context switching. The resulting system is easier to build and extend than a similar X system, because often changes need only be made in one place. Or in the case of protocol extensions, two (rather than 3 or 4 in the X case where window management and/or composition handling may also need to be updated).

Types of Compositors

Compositors come in different types, depending on which role they play in the overall architecture of the OS. For instance, a *system compositor* can be used for booting the system, handling multiple user switching, a possible console terminal emulator and so forth. A different compositor, a *session compositor* would provide the actual desktop environment. There are many ways for different types of compositors to co-exist.

In this section, we introduce three types of Wayland compositors relying on *libwayland-server*.

2.1. System Compositor

A system compositor can run from early boot until shutdown. It effectively replaces the kernel vt system, and can tie in with the systems graphical boot setup and multiseat support.

A system compositor can host different types of session compositors, and let us switch between multiple sessions (fast user switching, or secure/personal desktop switching).

A linux implementation of a system compositor will typically use libudev, egl, kms, evdev and cairo.

For fullscreen clients, the system compositor can reprogram the video scanout address to read directly from the client provided buffer.

2.2. Session Compositor

A session compositor is responsible for a single user session. If a system compositor is present, the session compositor will run nested under the system compositor. Nesting is feasible because the protocol is asynchronous; roundtrips would be too expensive when nesting is involved. If no system compositor is present, a session compositor can run directly on the hw.

X applications can continue working under a session compositor by means of a root-less X server that is activated on demand.

Possible examples for session compositors include

- gnome-shell
- moblin
- kwin
- kmscon
- rdp session
- Weston with X11 or Wayland backend is a session compositor nested in another session compositor.
- fullscreen X session under Wayland

2.3. Embedding Compositor

X11 lets clients embed windows from other clients, or lets clients copy pixmap contents rendered by another client into their window. This is often used for applets in a panel, browser plugins and similar. Wayland doesn't directly allow this, but clients can communicate GEM buffer names out-of-band, for example, using D-Bus, or command line arguments when the panel launches the applet. Another option is to use a nested Wayland instance. For this, the Wayland server will have to be a library that

the host application links to. The host application will then pass the Wayland server socket name to the embedded application, and will need to implement the Wayland compositor interface. The host application composites the client surfaces as part of it's window, that is, in the web page or in the panel. The benefit of nesting the Wayland server is that it provides the requests the embedded client needs to inform the host about buffer updates and a mechanism for forwarding input events from the host application.

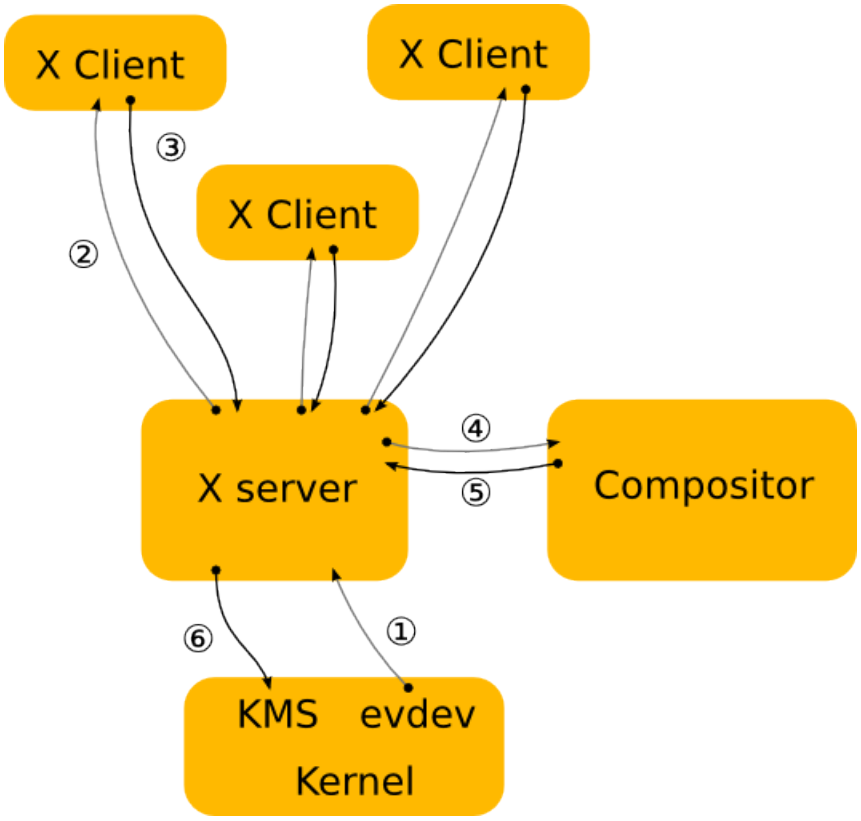
An example for this kind of setup is firefox embedding the flash player as a kind of special-purpose compositor.

Wayland Architecture

3.1. X vs. Wayland Architecture

A good way to understand the Wayland architecture and how it is different from X is to follow an event from the input device to the point where the change it affects appears on screen.

This is where we are now with X:



1. The kernel gets an event from an input device and sends it to X through the evdev input driver. The kernel does all the hard work here by driving the device and translating the different device specific event protocols to the linux evdev input event standard.

2. The X server determines which window the event affects and sends it to the clients that have selected for the event in question on that window. The X server doesn't actually know how to do this right, since the window location on screen is controlled by the compositor and may be transformed in a number of ways that the X server doesn't understand (scaled down, rotated, wobbling, etc).

3. The client looks at the event and decides what to do. Often the UI will have to change in response to the event - perhaps a check box was clicked or the pointer entered a button that must be highlighted. Thus the client sends a rendering request back to the X server.

4. When the X server receives the rendering request, it sends it to the driver to let it program the hardware to do the rendering. The X server also calculates the bounding region of the rendering, and sends that to the compositor as a damage event.

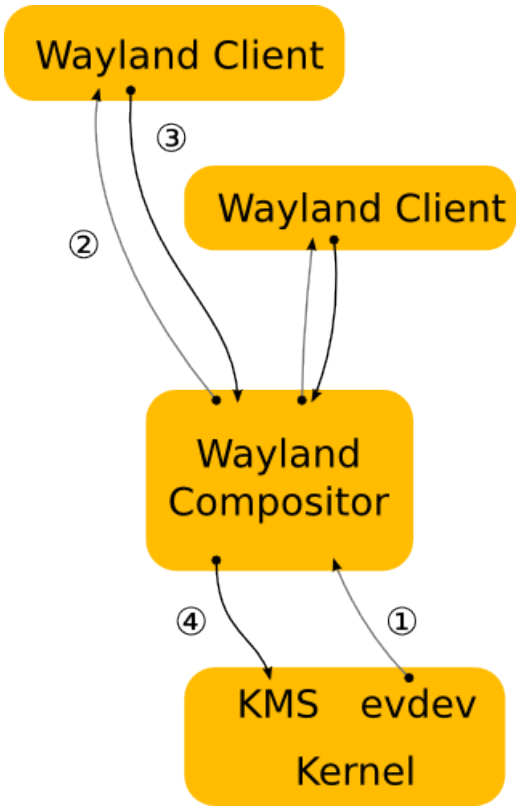
5. The damage event tells the compositor that something changed in the window and that it has to recomposite the part of the screen where that window is visible. The compositor is responsible for

rendering the entire screen contents based on its scenegraph and the contents of the X windows. Yet, it has to go through the X server to render this.

6. The X server receives the rendering requests from the compositor and either copies the compositor back buffer to the front buffer or does a pageflip. In the general case, the X server has to do this step so it can account for overlapping windows, which may require clipping and determine whether or not it can page flip. However, for a compositor, which is always fullscreen, this is another unnecessary context switch.

As suggested above, there are a few problems with this approach. The X server doesn't have the information to decide which window should receive the event, nor can it transform the screen coordinates to window local coordinates. And even though X has handed responsibility for the final painting of the screen to the compositing manager, X still controls the front buffer and modesetting. Most of the complexity that the X server used to handle is now available in the kernel or self contained libraries (KMS, evdev, mesa, fontconfig, freetype, cairo, Qt etc). In general, the X server is now just a middle man that introduces an extra step between applications and the compositor and an extra step between the compositor and the hardware.

In Wayland the compositor is the display server. We transfer the control of KMS and evdev to the compositor. The Wayland protocol lets the compositor send the input events directly to the clients and lets the client send the damage event directly to the compositor:



1. The kernel gets an event and sends it to the compositor. This is similar to the X case, which is great, since we get to reuse all the input drivers in the kernel.

2. The compositor looks through its scenegraph to determine which window should receive the event. The scenegraph corresponds to what's on screen and the compositor understands the transformations that it may have applied to the elements in the scenegraph. Thus, the compositor can pick the right window and transform the screen coordinates to window local coordinates, by applying the inverse transformations. The types of transformation that can be applied to a

window is only restricted to what the compositor can do, as long as it can compute the inverse transformation for the input events.

3. As in the X case, when the client receives the event, it updates the UI in response. But in the Wayland case, the rendering happens in the client, and the client just sends a request to the compositor to indicate the region that was updated.

4. The compositor collects damage requests from its clients and then recomposites the screen. The compositor can then directly issue an ioctl to schedule a pageflip with KMS.

3.2. Wayland Rendering

One of the details I left out in the above overview is how clients actually render under Wayland. By removing the X server from the picture we also removed the mechanism by which X clients typically render. But there's another mechanism that we're already using with DRI2 under X: direct rendering. With direct rendering, the client and the server share a video memory buffer. The client links to a rendering library such as OpenGL that knows how to program the hardware and renders directly into the buffer. The compositor in turn can take the buffer and use it as a texture when it composites the desktop. After the initial setup, the client only needs to tell the compositor which buffer to use and when and where it has rendered new content into it.

This leaves an application with two ways to update its window contents:

1. Render the new content into a new buffer and tell the compositor to use that instead of the old buffer. The application can allocate a new buffer every time it needs to update the window contents or it can keep two (or more) buffers around and cycle between them. The buffer management is entirely under application control.

2. Render the new content into the buffer that it previously told the compositor to to use. While it's possible to just render directly into the buffer shared with the compositor, this might race with the compositor. What can happen is that repainting the window contents could be interrupted by the compositor repainting the desktop. If the application gets interrupted just after clearing the window but before rendering the contents, the compositor will texture from a blank buffer. The result is that the application window will flicker between a blank window or half-rendered content. The traditional way to avoid this is to render the new content into a back buffer and then copy from there into the compositor surface. The back buffer can be allocated on the fly and just big enough to hold the new content, or the application can keep a buffer around. Again, this is under application control.

In either case, the application must tell the compositor which area of the surface holds new contents. When the application renders directly to the shared buffer, the compositor needs to be noticed that there is new content. But also when exchanging buffers, the compositor doesn't assume anything changed, and needs a request from the application before it will repaint the desktop. The idea that even if an application passes a new buffer to the compositor, only a small part of the buffer may be different, like a blinking cursor or a spinner.

3.3. Hardware Enabling for Wayland

Typically, hardware enabling includes modesetting/display and EGL/GLES2. On top of that Wayland needs a way to share buffers efficiently between processes. There are two sides to that, the client side and the server side.

On the client side we've defined a Wayland EGL platform. In the EGL model, that consists of the native types (EGLNativeDisplayType, EGLNativeWindowType and EGLNativePixmapType) and a way to create those types. In other words, it's the glue code that binds the EGL stack and its

buffer sharing mechanism to the generic Wayland API. The EGL stack is expected to provide an implementation of the Wayland EGL platform. The full API is in the wayland-egl.h header. The open source implementation in the mesa EGL stack is in wayland-egl.c and platform_wayland.c.

Under the hood, the EGL stack is expected to define a vendor-specific protocol extension that lets the client side EGL stack communicate buffer details with the compositor in order to share buffers. The point of the wayland-egl.h API is to abstract that away and just let the client create an EGLSurface for a Wayland surface and start rendering. The open source stack uses the drm Wayland extension, which lets the client discover the drm device to use and authenticate and then share drm (GEM) buffers with the compositor.

The server side of Wayland is the compositor and core UX for the vertical, typically integrating task switcher, app launcher, lock screen in one monolithic application. The server runs on top of a modesetting API (kernel modesetting, OpenWF Display or similar) and composites the final UI using a mix of EGL/GLES2 compositor and hardware overlays if available. Enabling modesetting, EGL/GLES2 and overlays is something that should be part of standard hardware bringup. The extra requirement for Wayland enabling is the EGL_WL_bind_wayland_display extension that lets the compositor create an EGLImage from a generic Wayland shared buffer. It's similar to the EGL_KHR_image_pixmap extension to create an EGLImage from an X pixmap.

The extension has a setup step where you have to bind the EGL display to a Wayland display. Then as the compositor receives generic Wayland buffers from the clients (typically when the client calls eglSwapBuffers), it will be able to pass the struct wl_buffer pointer to eglCreateImageKHR as the EGLClientBuffer argument and with EGL_WAYLAND_BUFFER_WL as the target. This will create an EGLImage, which can then be used by the compositor as a texture or passed to the modesetting code to use as an overlay plane. Again, this is implemented by the vendor specific protocol extension, which on the server side will receive the driver specific details about the shared buffer and turn that into an EGL image when the user calls eglCreateImageKHR.

Wayland Protocol and Model of Operation

4.1. Basic Principles

The Wayland protocol is an asynchronous object oriented protocol. All requests are method invocations on some object. The requests include an object ID that uniquely identifies an object on the server. Each object implements an interface and the requests include an opcode that identifies which method in the interface to invoke.

The server sends back events to the client, each event is emitted from an object. Events can be error conditions. The event includes the object ID and the event opcode, from which the client can determine the type of event. Events are generated both in response to requests (in which case the request and the event constitutes a round trip) or spontaneously when the server state changes.

- State is broadcast on connect, events are sent out when state changes. Clients must listen for these changes and cache the state. There is no need (or mechanism) to query server state.
- The server will broadcast the presence of a number of global objects, which in turn will broadcast their current state.

4.2. Code Generation

The interfaces, requests and events are defined in **`protocol/wayland.xml`**. This xml is used to generate the function prototypes that can be used by clients and compositors.

The protocol entry points are generated as inline functions which just wrap the `wl_proxy_*` functions. The inline functions aren't part of the library ABI and language bindings should generate their own stubs for the protocol entry points from the xml.

4.3. Wire Format

The protocol is sent over a UNIX domain stream socket, where the endpoint usually is named `wayland-0` (although it can be changed via *WAYLAND_DISPLAY* in the environment). The protocol is message-based. A message sent by a client to the server is called request. A message from the server to a client is called event. Every message is structured as 32-bit words, values are represented in the host's byte-order.

The message header has 2 words in it:

- The first word is the sender's object ID (32-bit).
- The second has 2 parts of 16-bit. The upper 16-bits are the message size in bytes, starting at the header (i.e. it has a minimum value of 8).The lower is the request/event opcode.

The payload describes the request/event arguments. Every argument is always aligned to 32-bits. Where padding is required, the value of padding bytes is undefined. There is no prefix that describes the type, but it is inferred implicitly from the xml specification.

The representation of argument types are as follows:

int, uint

The value is the 32-bit value of the signed/unsigned int.

fixed
: Signed 24.8 decimal numbers. It is a signed decimal type which offers a sign bit, 23 bits of integer precision and 8 bits of decimal precision. This is exposed as an opaque struct with conversion helpers to and from double and int on the C API side.

string
: Starts with an unsigned 32-bit length, followed by the string contents, including terminating null byte, then padding to a 32-bit boundary.

object
: 32-bit object ID.

new_id
: The 32-bit object ID. On requests, the client decides the ID. The only events with new_id are advertisements of globals, and the server will use IDs below 0x10000.

array
: Starts with 32-bit array size in bytes, followed by the array contents verbatim, and finally padding to a 32-bit boundary.

fd
: The file descriptor is not stored in the message buffer, but in the ancillary data of the UNIX domain socket message (msg_control).

4.4. Interfaces

The protocol includes several interfaces which are used for interacting with the server. Each interface provides requests, events, and errors (which are really just special events) as described above. Specific compositor implementations may have their own interfaces provided as extensions, but there are several which are always expected to be present.

Core interfaces:

wl_display - core global object
: The core global object. This is a special singleton object. It is used for internal Wayland protocol features.

wl_registry - global registry object
: The global registry object. The server has a number of global objects that are available to all clients. These objects typically represent an actual object in the server (for example, an input device) or they are singleton objects that provide extension functionality. When a client creates a registry object, the registry object will emit a global event for each global currently in the registry. Globals come and go as a result of device or monitor hotplugs, reconfiguration or other events, and the registry will send out global and global_remove events to keep the client up to date with the changes. To mark the end of the initial burst of events, the client can use the wl_display.sync request immediately after calling wl_display.get_registry. A client can bind to a global object by using the bind request. This creates a client-side handle that lets the object emit events to the client and lets the client invoke requests on the object.

wl_callback - callback object
: Clients can handle the 'done' event to get notified when the related request is done.

wl_compositor - the compositor singleton
: A compositor. This object is a singleton global. The compositor is in charge of combining the contents of multiple surfaces into one displayable output.

wl_shm_pool - a shared memory pool

The wl_shm_pool object encapsulates a piece of memory shared between the compositor and client. Through the wl_shm_pool object, the client can allocate shared memory wl_buffer objects. All objects created through the same pool share the same underlying mapped memory. Reusing the mapped memory avoids the setup/teardown overhead and is useful when interactively resizing a surface or for many small buffers.

wl_shm - shared memory support

A global singleton object that provides support for shared memory. Clients can create wl_shm_pool objects using the create_pool request. At connection setup time, the wl_shm object emits one or more format events to inform clients about the valid pixel formats that can be used for buffers.

wl_buffer - content for a wl_surface

A buffer provides the content for a wl_surface. Buffers are created through factory interfaces such as wl_drm, wl_shm or similar. It has a width and a height and can be attached to a wl_surface, but the mechanism by which a client provides and updates the contents is defined by the buffer factory interface.

wl_data_offer - offer to transfer data

A wl_data_offer represents a piece of data offered for transfer by another client (the source client). It is used by the copy-and-paste and drag-and-drop mechanisms. The offer describes the different mime types that the data can be converted to and provides the mechanism for transferring the data directly from the source client.

wl_data_source - offer to transfer data

The wl_data_source object is the source side of a wl_data_offer. It is created by the source client in a data transfer and provides a way to describe the offered data and a way to respond to requests to transfer the data.

wl_data_device - data transfer device

There is one wl_data_device per seat which can be obtained from the global wl_data_device_manager singleton. A wl_data_device provides access to inter-client data transfer mechanisms such as copy-and-paste and drag-and-drop.

wl_data_device_manager - data transfer interface

The wl_data_device_manager is a singleton global object that provides access to inter-client data transfer mechanisms such as copy-and-paste and drag-and-drop. These mechanisms are tied to a wl_seat and this interface lets a client get a wl_data_device corresponding to a wl_seat.

wl_shell - create desktop-style surfaces

This interface is implemented by servers that provide desktop-style user interfaces. It allows clients to associate a wl_shell_surface with a basic surface.

wl_shell_surface - desktop-style metadata interface

An interface that may be implemented by a wl_surface, for implementations that provide a desktop-style user interface. It provides requests to treat surfaces like toplevel, fullscreen or popup windows, move, resize or maximize them, associate metadata like title and class, etc. On the server side the object is automatically destroyed when the related wl_surface is destroyed. On client side, wl_shell_surface_destroy() must be called before destroying the wl_surface object.

wl_surface - an onscreen surface

A surface is a rectangular area that is displayed on the screen. It has a location, size and pixel contents. The size of a surface (and relative positions on it) is described in surface local coordinates, which may differ from the buffer local coordinates of the pixel content, in case a

buffer_transform or a buffer_scale is used. Surfaces are also used for some special purposes, e.g. as cursor images for pointers, drag icons, etc.

wl_seat - group of input devices
: A seat is a group of keyboards, pointer and touch devices. This object is published as a global during start up, or when such a device is hot plugged. A seat typically has a pointer and maintains a keyboard focus and a pointer focus.

wl_pointer - pointer input device
: The wl_pointer interface represents one or more input devices, such as mice, which control the pointer location and pointer_focus of a seat. The wl_pointer interface generates motion, enter and leave events for the surfaces that the pointer is located over, and button and axis events for button presses, button releases and scrolling.

wl_keyboard - keyboard input device
: The wl_keyboard interface represents one or more keyboards associated with a seat.

wl_touch - touchscreen input device
: The wl_touch interface represents a touchscreen associated with a seat. Touch interactions can consist of one or more contacts. For each contact, a series of events is generated, starting with a down event, followed by zero or more motion events, and ending with an up event. Events relating to the same contact point can be identified by the ID of the sequence.

wl_output - compositor output region
: An output describes part of the compositor geometry. The compositor works in the 'compositor coordinate system' and an output corresponds to rectangular area in that space that is actually visible. This typically corresponds to a monitor that displays part of the compositor space. This object is published as global during start up, or when a monitor is hotplugged.

wl_region - region interface
: A region object describes an area. Region objects are used to describe the opaque and input regions of a surface.

4.5. Versioning

Every interface is versioned and every protocol object implements a particular version of its interface. For global objects, the maximum version supported by the server is advertised with the global and the actual verion of the created protocol object is determined by the version argument passed to wl_registry.bind(). For objects that are not globals, their version is inferred from the object that created them.

In order to keep things sane, this has a few implications for interface versions:

- The object creation hierarchy must be a tree. Otherwise, infering object versions from the parent object becomes a much more difficult to properly track.
- When the version of an interface increases, so does the version of its parent (recursively until you get to a global interface)
- A global interface's version number acts like a counter for all of its child interfaces. Whenever a child interface gets modified, the global parent's interface version number also increases (see above). The child interface then takes on the same version number as the new version of its parent global interface.

To illustrate the above, consider the wl_compositor interface. It has two children, wl_surface and wl_region. As of wayland version 1.2, wl_surface and wl_compositor are both at version 3. If

something is added to the wl_region interface, both wl_region and wl_compositor will get bumpped to version 4. If, afterwards, wl_surface is changed, both wl_compositor and wl_surface will be at version 5. In this way the global interface version is used as a sort of "counter" for all of its child interfaces. This makes it very simple to know the version of the child given the version of its parent. The child is at the highest possible interface version that is less than or equal to its parent's version.

It is worth noting a particular exception to the above versioning scheme. The wl_display (and, by extension, wl_registry) interface cannot change because it is the core protocol object and its version is never advertised nor is there a mechanism to request a different version.

4.6. Connect Time

There is no fixed connection setup information, the server emits multiple events at connect time, to indicate the presence and properties of global objects: outputs, compositor, input devices.

4.7. Security and Authentication

- mostly about access to underlying buffers, need new drm auth mechanism (the grant-to ioctl idea), need to check the cmd stream?
- getting the server socket depends on the compositor type, could be a system wide name, through fd passing on the session dbus. or the client is forked by the compositor and the fd is already opened.

4.8. Creating Objects

Each object has a unique ID. The IDs are allocated by the entity creating the object (either client or server). IDs allocated by the client are in the range [1, 0xefffffff] while IDs allocated by the server are in the range [0xff000000, 0xffffffff]. The 0 ID is reserved to represent a null or non-existant object. For efficiency purposes, the IDs are densely packed in the sense that the ID N will not be used until N-1 has been used. Any ID allocation algorithm that does not maintain this property is incompatible with the implementation in libwayland.

4.9. Compositor

The compositor is a global object, advertised at connect time.

See *Section A.4, “wl_compositor - the compositor singleton”* for the protocol description.

4.10. Surfaces

Surfaces are created by the client. Clients don't know the global position of their surfaces, and cannot access other clients surfaces.

See *Section A.14, “wl_surface - an onscreen surface”* for the protocol description.

4.11. Input

A seat represents a group of input devices including mice, keyboards and touchscreens. It has a keyboard and pointer focus. Seats are global objects. Pointer events are delivered in surface local coordinates.

The compositor maintains an implicit grab when a button is pressed, to ensure that the corresponding button release event gets delivered to the same surface. But there is no way for clients to take

an explicit grab. Instead, surfaces can be mapped as 'popup', which combines transient window semantics with a pointer grab.

To avoid race conditions, input events that are likely to trigger further requests (such as button presses, key events, pointer motions) carry serial numbers, and requests such as wl_surface.set_popup require that the serial number of the triggering event is specified. The server maintains a monotonically increasing counter for these serial numbers.

Input events also carry timestamps with millisecond granularity. Their base is undefined, so they can't be compared against system time (as obtained with clock_gettime or gettimeofday). They can be compared with each other though, and for instance be used to identify sequences of button presses as double or triple clicks.

See *Section A.15, "wl_seat - group of input devices"* for the protocol description.

Talk about:

- keyboard map, change events
- xkb on Wayland
- multi pointer Wayland

A surface can change the pointer image when the surface is the pointer focus of the input device. Wayland doesn't automatically change the pointer image when a pointer enters a surface, but expects the application to set the cursor it wants in response to the pointer focus and motion events. The rationale is that a client has to manage changing pointer images for UI elements within the surface in response to motion events anyway, so we'll make that the only mechanism for setting or changing the pointer image. If the server receives a request to set the pointer image after the surface loses pointer focus, the request is ignored. To the client this will look like it successfully set the pointer image.

The compositor will revert the pointer image back to a default image when no surface has the pointer focus for that device. Clients can revert the pointer image back to the default image by setting a NULL image.

What if the pointer moves from one window which has set a special pointer image to a surface that doesn't set an image in response to the motion event? The new surface will be stuck with the special pointer image. We can't just revert the pointer image on leaving a surface, since if we immediately enter a surface that sets a different image, the image will flicker. Broken app, I suppose.

4.12. Output

An output is a global object, advertised at connect time or as it comes and goes.

See *Section A.19, "wl_output - compositor output region"* for the protocol description.

- laid out in a big (compositor) coordinate system
- basically xrandr over Wayland
- geometry needs position in compositor coordinate system
- events to advertise available modes, requests to move and change modes

4.13. Data sharing between clients

The Wayland protocol provides clients a mechanism for sharing data that allows the implementation of copy-paste and drag-and-drop. The client providing the data creates a `wl_data_source` object and

the clients obtaining the data will see it as `wl_data_offer` object. This interface allows the clients to agree on a mutually supported mime type and transfer the data via a file descriptor that is passed through the protocol.

The next section explains the negotiation between data source and data offer objects. *Section 4.13.2, "Data devices"* explains how these objects are created and passed to different clients using the `wl_data_device` interface that implements copy-paste and drag-and-drop support.

See *Section A.8, "wl_data_offer - offer to transfer data"*, *Section A.9, "wl_data_source - offer to transfer data"*, *Section A.10, "wl_data_device - data transfer device"* and *Section A.11, "wl_data_device_manager - data transfer interface"* for protocol descriptions.

MIME is defined in RFC's 2045-2049. A *registry of MIME types*[1] is maintained by the Internet Assigned Numbers Authority (IANA).

4.13.1. Data negotiation

A client providing data to other clients will create a `wl_data_source` object and advertise the mime types for the formats it supports for that data through the `wl_data_source.offer` request. On the receiving end, the data offer object will generate one `wl_data_offer.offer` event for each supported mime type.

The actual data transfer happens when the receiving client sends a `wl_data_offer.receive` request. This request takes a mime type and a file descriptor as arguments. This request will generate a `wl_data_source.send` event on the sending client with the same arguments, and the latter client is expected to write its data to the given file descriptor using the chosen mime type.

4.13.2. Data devices

Data devices glue data sources and offers together. A data device is associated with a `wl_seat` and is obtained by the clients using the `wl_data_device_manager` factory object, which is also responsible for creating data sources.

Clients are informed of new data offers through the `wl_data_device.data_offer` event. After this event is generated the data offer will advertise the available mime types. New data offers are introduced prior to their use for copy-paste or drag-and-drop.

4.13.2.1. Selection

Each data device has a selection data source. Clients create a data source object using the device manager and may set it as the current selection for a given data device. Whenever the current selection changes, the client with keyboard focus receives a `wl_data_device.selection` event. This event is also generated on a client immediately before it receives keyboard focus.

The data offer is introduced with `wl_data_device.data_offer` event before the selection event.

4.13.2.2. Drag and Drop

A drag-and-drop operation is started using the `wl_data_device.start_drag` request. This requests causes a pointer grab that will generate enter, motion and leave events on the data device. A data source is supplied as argument to start_drag, and data offers associated with it are supplied to clients surfaces under the pointer in the `wl_data_device.enter` event. The data offer is introduced to the client prior to the enter event with the `wl_data_device.data_offer` event.

[1] ftp://ftp.isi.edu/in-notes/iana/assignments/media-types/

Clients are expected to provide feedback to the data sending client by calling the `wl_data_offer.accept` request with a mime type it accepts. If none of the advertised mime types is supported by the receiving client, it should supply NULL to the accept request. The accept request causes the sending client to receive a `wl_data_source.target` event with the chosen mime type.

When the drag ends, the receiving client receives a `wl_data_device.drop` event at which it is expected to transfer the data using the `wl_data_offer.receive` request.

Appendix A. Wayland Protocol Specification

A.1. wl_display - core global object

The core global object. This is a special singleton object. It is used for internal Wayland protocol features.

A.1.1. Requests provided by wl_display

A.1.1.1. wl_display::sync - asynchronous roundtrip

The sync request asks the server to emit the 'done' event on the returned wl_callback object. Since requests are handled in-order and events are delivered in-order, this can be used as a barrier to ensure all previous requests and the resulting events have been handled.

The object returned by this request will be destroyed by the compositor after the callback is fired and as such the client must not attempt to use it after that point.

wl_display::sync arguments

callback

Type: new_id

A.1.1.2. wl_display::get_registry - get global registry object

This request creates a registry object that allows the client to list and bind the global objects available from the compositor.

wl_display::get_registry arguments

callback

Type: new_id

A.1.2. Events provided by wl_display

A.1.2.1. wl_display::error - fatal error event

The error event is sent out when a fatal (non-recoverable) error has occurred. The object_id argument is the object where the error occurred, most often in response to a request to that object. The code identifies the error and is defined by the object interface. As such, each interface defines its own set of error codes. The message is an brief description of the error, for (debugging) convenience.

wl_display::error arguments

object_id
: Type: object

code
: Type: uint

message
: Type: string

A.1.2.2. wl_display::delete_id - acknowledge object ID deletion

This event is used internally by the object ID management logic. When a client deletes an object, the server will send this event to acknowledge that it has seen the delete request. When the client receive this event, it will know that it can safely reuse the object ID.

wl_display::delete_id arguments

id
: Type: uint

A.1.3. Enums provided by wl_display

A.1.3.1. wl_display::error - global error values

These errors are global and can be emitted in response to any server request.

wl_display::error values

invalid_object
: Value: 0

  server couldn't find object

invalid_method
: Value: 1

  method doesn't exist on the specified interface

no_memory
: Value: 2

  server is out of memory

A.2. wl_registry - global registry object

The global registry object. The server has a number of global objects that are available to all clients. These objects typically represent an actual object in the server (for example, an input device) or they are singleton objects that provide extension functionality.

When a client creates a registry object, the registry object will emit a global event for each global currently in the registry. Globals come and go as a result of device or monitor hotplugs, reconfiguration or other events, and the registry will send out global and global_remove events to keep the client up to date with the changes. To mark the end of the initial burst of events, the client can use the wl_display.sync request immediately after calling wl_display.get_registry.

A client can bind to a global object by using the bind request. This creates a client-side handle that lets the object emit events to the client and lets the client invoke requests on the object.

A.2.1. Requests provided by wl_registry

A.2.1.1. wl_registry::bind - bind an object to the display

Binds a new, client-created object to the server using the specified name as the identifier.

wl_registry::bind arguments

name

Type: uint

unique name for the object

id

Type: new_id

A.2.2. Events provided by wl_registry

A.2.2.1. wl_registry::global - announce global object

Notify the client of global objects.

The event notifies the client that a global object with the given name is now available, and it implements the given version of the given interface.

wl_registry::global arguments

name

Type: uint

interface

Type: string

version

Type: uint

A.2.2.2. wl_registry::global_remove - announce removal of global object

Notify the client of removed global objects.

This event notifies the client that the global identified by name is no longer available. If the client bound to the global using the bind request, the client should now destroy that object.

The object remains valid and requests to the object will be ignored until the client destroys it, to avoid races between the global going away and a client sending a request to it.

wl_registry::global_remove arguments
name
Type: uint

A.3. wl_callback - callback object

Clients can handle the 'done' event to get notified when the related request is done.

A.3.1. Events provided by wl_callback

A.3.1.1. wl_callback::done - done event

Notify the client when the related request is done.

wl_callback::done arguments
serial
Type: uint

serial of the event

A.4. wl_compositor - the compositor singleton

A compositor. This object is a singleton global. The compositor is in charge of combining the contents of multiple surfaces into one displayable output.

A.4.1. Requests provided by wl_compositor

A.4.1.1. wl_compositor::create_surface - create new surface

Ask the compositor to create a new surface.

wl_compositor::create_surface arguments
id
Type: new_id

A.4.1.2. wl_compositor::create_region - create new region

Ask the compositor to create a new region.

wl_compositor::create_region arguments
id
Type: new_id

A.5. wl_shm_pool - a shared memory pool

The wl_shm_pool object encapsulates a piece of memory shared between the compositor and client. Through the wl_shm_pool object, the client can allocate shared memory wl_buffer objects. All objects created through the same pool share the same underlying mapped memory. Reusing the mapped

memory avoids the setup/teardown overhead and is useful when interactively resizing a surface or for many small buffers.

A.5.1. Requests provided by wl_shm_pool

A.5.1.1. wl_shm_pool::create_buffer - create a buffer from the pool

Create a wl_buffer object from the pool.

The buffer is created offset bytes into the pool and has width and height as specified. The stride arguments specifies the number of bytes from beginning of one row to the beginning of the next. The format is the pixel format of the buffer and must be one of those advertised through the wl_shm.format event.

A buffer will keep a reference to the pool it was created from so it is valid to destroy the pool immediately after creating a buffer from it.

wl_shm_pool::create_buffer arguments

id
: Type: new_id

offset
: Type: int

width
: Type: int

height
: Type: int

stride
: Type: int

format
: Type: uint

A.5.1.2. wl_shm_pool::destroy - destroy the pool

Destroy the shared memory pool.

The mmapped memory will be released when all buffers that have been created from this pool are gone.

A.5.1.3. wl_shm_pool::resize - change the size of the pool mapping

This request will cause the server to remap the backing memory for the pool from the file descriptor passed when the pool was created, but using the new size.

wl_shm_pool::resize arguments

size
: Type: int

A.6. wl_shm - shared memory support

A global singleton object that provides support for shared memory.

Clients can create wl_shm_pool objects using the create_pool request.

At connection setup time, the wl_shm object emits one or more format events to inform clients about the valid pixel formats that can be used for buffers.

A.6.1. Requests provided by wl_shm

A.6.1.1. wl_shm::create_pool - create a shm pool

Create a new wl_shm_pool object.

The pool can be used to create shared memory based buffer objects. The server will mmap size bytes of the passed file descriptor, to use as backing memory for the pool.

wl_shm::create_pool arguments

id

Type: new_id

fd

Type: fd

size

Type: int

A.6.2. Events provided by wl_shm

A.6.2.1. wl_shm::format - pixel format description

Informs the client about a valid pixel format that can be used for buffers. Known formats include argb8888 and xrgb8888.

wl_shm::format arguments

format

Type: uint

A.6.3. Enums provided by wl_shm

A.6.3.1. wl_shm::error - wl_shm error values

These errors can be emitted in response to wl_shm requests.

wl_shm::error values

invalid_format

Value: 0

buffer format is not known

invalid_stride

Value: 1

invalid size or stride during pool or buffer creation

invalid_fd

Value: 2

mmapping the file descriptor failed

A.6.3.2. wl_shm::format - pixel formats

This describes the memory layout of an individual pixel.

All renderers should support argb8888 and xrgb8888 but any other formats are optional and may not be supported by the particular renderer in use.

wl_shm::format values

argb8888
: Value: 0

  32-bit ARGB format

xrgb8888
: Value: 1

  32-bit RGB format

c8
: Value: 0x20203843

rgb332
: Value: 0x38424752

bgr233
: Value: 0x38524742

xrgb4444
: Value: 0x32315258

xbgr4444
: Value: 0x32314258

rgbx4444
: Value: 0x32315852

bgrx4444
: Value: 0x32315842

argb4444
: Value: 0x32315241

abgr4444
: Value: 0x32314241

rgba4444
: Value: 0x32314152

bgra4444
: Value: 0x32314142

xrgb1555
: Value: 0x35315258

xbgr1555
: Value: 0x35314258

rgbx5551
: Value: 0x35315852

bgrx5551
: Value: 0x35315842

argb1555
: Value: 0x35315241

abgr1555
: Value: 0x35314241

rgba5551
: Value: 0x35314152

bgra5551
: Value: 0x35314142

rgb565
: Value: 0x36314752

bgr565
: Value: 0x36314742

rgb888
: Value: 0x34324752

bgr888
: Value: 0x34324742

xbgr8888
: Value: 0x34324258

rgbx8888
: Value: 0x34325852

bgrx8888
: Value: 0x34325842

abgr8888
: Value: 0x34324241

rgba8888
: Value: 0x34324152

bgra8888
: Value: 0x34324142

xrgb2101010
: Value: 0x30335258

xbgr2101010
: Value: 0x30334258

rgbx1010102
Value: 0x30335852

bgrx1010102
Value: 0x30335842

argb2101010
Value: 0x30335241

abgr2101010
Value: 0x30334241

rgba1010102
Value: 0x30334152

bgra1010102
Value: 0x30334142

yuyv
Value: 0x56595559

yvyu
Value: 0x55595659

uyvy
Value: 0x59565955

vyuy
Value: 0x59555956

ayuv
Value: 0x56555941

nv12
Value: 0x3231564e

nv21
Value: 0x3132564e

nv16
Value: 0x3631564e

nv61
Value: 0x3136564e

yuv410
Value: 0x39565559

yvu410
Value: 0x39555659

yuv411
Value: 0x31315559

yvu411
Value: 0x31315659

yuv420
: Value: 0x32315559

yvu420
: Value: 0x32315659

yuv422
: Value: 0x36315559

yvu422
: Value: 0x36315659

yuv444
: Value: 0x34325559

yvu444
: Value: 0x34325659

A.7. wl_buffer - content for a wl_surface

A buffer provides the content for a wl_surface. Buffers are created through factory interfaces such as wl_drm, wl_shm or similar. It has a width and a height and can be attached to a wl_surface, but the mechanism by which a client provides and updates the contents is defined by the buffer factory interface.

A.7.1. Requests provided by wl_buffer

A.7.1.1. wl_buffer::destroy - destroy a buffer

Destroy a buffer. If and how you need to release the backing storage is defined by the buffer factory interface.

For possible side-effects to a surface, see wl_surface.attach.

A.7.2. Events provided by wl_buffer

A.7.2.1. wl_buffer::release - compositor releases buffer

Sent when this wl_buffer is no longer used by the compositor. The client is now free to re-use or destroy this buffer and its backing storage.

If a client receives a release event before the frame callback requested in the same wl_surface.commit that attaches this wl_buffer to a surface, then the client is immediately free to re-use the buffer and its backing storage, and does not need a second buffer for the next surface content update. Typically this is possible, when the compositor maintains a copy of the wl_surface contents, e.g. as a GL texture. This is an important optimization for GL(ES) compositors with wl_shm clients.

A.8. wl_data_offer - offer to transfer data

A wl_data_offer represents a piece of data offered for transfer by another client (the source client). It is used by the copy-and-paste and drag-and-drop mechanisms. The offer describes the different mime types that the data can be converted to and provides the mechanism for transferring the data directly from the source client.

A.8.1. Requests provided by wl_data_offer

A.8.1.1. wl_data_offer::accept - accept one of the offered mime types

Indicate that the client can accept the given mime type, or NULL for not accepted.

Used for feedback during drag-and-drop.

wl_data_offer::accept arguments

serial
: Type: uint

mime_type
: Type: string

A.8.1.2. wl_data_offer::receive - request that the data is transferred

To transfer the offered data, the client issues this request and indicates the mime type it wants to receive. The transfer happens through the passed file descriptor (typically created with the pipe system call). The source client writes the data in the mime type representation requested and then closes the file descriptor.

The receiving client reads from the read end of the pipe until EOF and the closes its end, at which point the transfer is complete.

wl_data_offer::receive arguments

mime_type
: Type: string

fd
: Type: fd

A.8.1.3. wl_data_offer::destroy - destroy data offer

Destroy the data offer.

A.8.2. Events provided by wl_data_offer

A.8.2.1. wl_data_offer::offer - advertise offered mime type

Sent immediately after creating the wl_data_offer object. One event per offered mime type.

wl_data_offer::offer arguments

mime_type
: Type: string

A.9. wl_data_source - offer to transfer data

The wl_data_source object is the source side of a wl_data_offer. It is created by the source client in a data transfer and provides a way to describe the offered data and a way to respond to requests to transfer the data.

A.9.1. Requests provided by wl_data_source

A.9.1.1. wl_data_source::offer - add an offered mime type

This request adds a mime type to the set of mime types advertised to targets. Can be called several times to offer multiple types.

wl_data_source::offer arguments

mime_type

Type: string

A.9.1.2. wl_data_source::destroy - destroy the data source

Destroy the data source.

A.9.2. Events provided by wl_data_source

A.9.2.1. wl_data_source::target - a target accepts an offered mime type

Sent when a target accepts pointer_focus or motion events. If a target does not accept any of the offered types, type is NULL.

Used for feedback during drag-and-drop.

wl_data_source::target arguments

mime_type

Type: string

A.9.2.2. wl_data_source::send - send the data

Request for data from the client. Send the data as the specified mime type over the passed file descriptor, then close it.

wl_data_source::send arguments

mime_type

Type: string

fd

Type: fd

A.9.2.3. wl_data_source::cancelled - selection was cancelled

This data source has been replaced by another data source. The client should clean up and destroy this data source.

A.10. wl_data_device - data transfer device

There is one wl_data_device per seat which can be obtained from the global wl_data_device_manager singleton.

A wl_data_device provides access to inter-client data transfer mechanisms such as copy-and-paste and drag-and-drop.

A.10.1. Requests provided by wl_data_device

A.10.1.1. wl_data_device::start_drag - start drag-and-drop operation

This request asks the compositor to start a drag-and-drop operation on behalf of the client.

The source argument is the data source that provides the data for the eventual data transfer. If source is NULL, enter, leave and motion events are sent only to the client that initiated the drag and the client is expected to handle the data passing internally.

The origin surface is the surface where the drag originates and the client must have an active implicit grab that matches the serial.

The icon surface is an optional (can be NULL) surface that provides an icon to be moved around with the cursor. Initially, the top-left corner of the icon surface is placed at the cursor hotspot, but subsequent wl_surface.attach request can move the relative position. Attach requests must be confirmed with wl_surface.commit as usual.

The current and pending input regions of the icon wl_surface are cleared, and wl_surface.set_input_region is ignored until the wl_surface is no longer used as the icon surface. When the use as an icon ends, the the current and pending input regions become undefined, and the wl_surface is unmapped.

wl_data_device::start_drag arguments

source
: Type: object

origin
: Type: object

icon
: Type: object

serial
: Type: uint

  serial of the implicit grab on the origin

A.10.1.2. wl_data_device::set_selection - copy data to the selection

This request asks the compositor to set the selection to the data from the source on behalf of the client.

To unset the selection, set the source to NULL.

wl_data_device::set_selection arguments

source
: Type: object

serial
: Type: uint

  serial of the event that triggered this request

A.10.2. Events provided by wl_data_device

A.10.2.1. wl_data_device::data_offer - introduce a new wl_data_offer

The data_offer event introduces a new wl_data_offer object, which will subsequently be used in either the data_device.enter event (for drag-and-drop) or the data_device.selection event (for selections). Immediately following the data_device_data_offer event, the new data_offer object will send out data_offer.offer events to describe the mime types it offers.

wl_data_device::data_offer arguments

id

Type: new_id

A.10.2.2. wl_data_device::enter - initiate drag-and-drop session

This event is sent when an active drag-and-drop pointer enters a surface owned by the client. The position of the pointer at enter time is provided by the x and y arguments, in surface local coordinates.

wl_data_device::enter arguments

serial

Type: uint

surface

Type: object

x

Type: fixed

y

Type: fixed

id

Type: object

A.10.2.3. wl_data_device::leave - end drag-and-drop session

This event is sent when the drag-and-drop pointer leaves the surface and the session ends. The client must destroy the wl_data_offer introduced at enter time at this point.

A.10.2.4. wl_data_device::motion - drag-and-drop session motion

This event is sent when the drag-and-drop pointer moves within the currently focused surface. The new position of the pointer is provided by the x and y arguments, in surface local coordinates.

wl_data_device::motion arguments

time

Type: uint

timestamp with millisecond granularity

x

Type: fixed

y

Type: fixed

A.10.2.5. wl_data_device::drop - end drag-and-drag session successfully

The event is sent when a drag-and-drop operation is ended because the implicit grab is removed.

A.10.2.6. wl_data_device::selection - advertise new selection

The selection event is sent out to notify the client of a new wl_data_offer for the selection for this device. The data_device.data_offer and the data_offer.offer events are sent out immediately before this event to introduce the data offer object. The selection event is sent to a client immediately before receiving keyboard focus and when a new selection is set while the client has keyboard focus. The data_offer is valid until a new data_offer or NULL is received or until the client loses keyboard focus.

wl_data_device::selection arguments

id

Type: object

A.11. wl_data_device_manager - data transfer interface

The wl_data_device_manager is a singleton global object that provides access to inter-client data transfer mechanisms such as copy-and-paste and drag-and-drop. These mechanisms are tied to a wl_seat and this interface lets a client get a wl_data_device corresponding to a wl_seat.

A.11.1. Requests provided by wl_data_device_manager

A.11.1.1. wl_data_device_manager::create_data_source - create a new data source

Create a new data source.

wl_data_device_manager::create_data_source arguments

id

Type: new_id

A.11.1.2. wl_data_device_manager::get_data_device - create a new data device

Create a new data device for a given seat.

wl_data_device_manager::get_data_device arguments

id

Type: new_id

seat

Type: object

A.12. wl_shell - create desktop-style surfaces

This interface is implemented by servers that provide desktop-style user interfaces.

It allows clients to associate a wl_shell_surface with a basic surface.

A.12.1. Requests provided by wl_shell

A.12.1.1. wl_shell::get_shell_surface - create a shell surface from a surface

Create a shell surface for an existing surface.

Only one shell surface can be associated with a given surface.

wl_shell::get_shell_surface arguments

id

Type: new_id

surface

Type: object

A.13. wl_shell_surface - desktop-style metadata interface

An interface that may be implemented by a wl_surface, for implementations that provide a desktop-style user interface.

It provides requests to treat surfaces like toplevel, fullscreen or popup windows, move, resize or maximize them, associate metadata like title and class, etc.

On the server side the object is automatically destroyed when the related wl_surface is destroyed. On client side, wl_shell_surface_destroy() must be called before destroying the wl_surface object.

A.13.1. Requests provided by wl_shell_surface

A.13.1.1. wl_shell_surface::pong - respond to a ping event

A client must respond to a ping event with a pong request or the client may be deemed unresponsive.

wl_shell_surface::pong arguments

serial

Type: uint

serial of the ping event

A.13.1.2. wl_shell_surface::move - start an interactive move

Start a pointer-driven move of the surface.

This request must be used in response to a button press event. The server may ignore move requests depending on the state of the surface (e.g. fullscreen or maximized).

wl_shell_surface::move arguments

seat

Type: object

the wl_seat whose pointer is used

serial

Type: uint

serial of the implicit grab on the pointer

A.13.1.3. wl_shell_surface::resize - start an interactive resize

Start a pointer-driven resizing of the surface.

This request must be used in response to a button press event. The server may ignore resize requests depending on the state of the surface (e.g. fullscreen or maximized).

wl_shell_surface::resize arguments

seat

Type: object

the wl_seat whose pointer is used

serial

Type: uint

serial of the implicit grab on the pointer

edges

Type: uint

which edge or corner is being dragged

A.13.1.4. wl_shell_surface::set_toplevel - make the surface a toplevel surface

Map the surface as a toplevel surface.

A toplevel surface is not fullscreen, maximized or transient.

A.13.1.5. wl_shell_surface::set_transient - make the surface a transient surface

Map the surface relative to an existing surface.

The x and y arguments specify the locations of the upper left corner of the surface relative to the upper left corner of the parent surface, in surface local coordinates.

The flags argument controls details of the transient behaviour.

wl_shell_surface::set_transient arguments

parent

Type: object

x

Type: int

y

Type: int

flags

Type: uint

A.13.1.6. wl_shell_surface::set_fullscreen - make the surface a fullscreen surface

Map the surface as a fullscreen surface.

If an output parameter is given then the surface will be made fullscreen on that output. If the client does not specify the output then the compositor will apply its policy - usually choosing the output on which the surface has the biggest surface area.

The client may specify a method to resolve a size conflict between the output size and the surface size - this is provided through the method parameter.

The framerate parameter is used only when the method is set to "driver", to indicate the preferred framerate. A value of 0 indicates that the app does not care about framerate. The framerate is specified in mHz, that is framerate of 60000 is 60Hz.

A method of "scale" or "driver" implies a scaling operation of the surface, either via a direct scaling operation or a change of the output mode. This will override any kind of output scaling, so that mapping a surface with a buffer size equal to the mode can fill the screen independent of buffer_scale.

A method of "fill" means we don't scale up the buffer, however any output scale is applied. This means that you may run into an edge case where the application maps a buffer with the same size of the output mode but buffer_scale 1 (thus making a surface larger than the output). In this case it is allowed to downscale the results to fit the screen.

The compositor must reply to this request with a configure event with the dimensions for the output on which the surface will be made fullscreen.

wl_shell_surface::set_fullscreen arguments

method
: Type: uint

framerate
: Type: uint

output
: Type: object

A.13.1.7. wl_shell_surface::set_popup - make the surface a popup surface

Map the surface as a popup.

A popup surface is a transient surface with an added pointer grab.

An existing implicit grab will be changed to owner-events mode, and the popup grab will continue after the implicit grab ends (i.e. releasing the mouse button does not cause the popup to be unmapped).

The popup grab continues until the window is destroyed or a mouse button is pressed in any other clients window. A click in any of the clients surfaces is reported as normal, however, clicks in other clients surfaces will be discarded and trigger the callback.

The x and y arguments specify the locations of the upper left corner of the surface relative to the upper left corner of the parent surface, in surface local coordinates.

wl_shell_surface::set_popup arguments

seat
: Type: object

  the wl_seat whose pointer is used

serial
: Type: uint

serial of the implicit grab on the pointer

parent
Type: object

x
Type: int

y
Type: int

flags
Type: uint

A.13.1.8. wl_shell_surface::set_maximized - make the surface a maximized surface

Map the surface as a maximized surface.

If an output parameter is given then the surface will be maximized on that output. If the client does not specify the output then the compositor will apply its policy - usually choosing the output on which the surface has the biggest surface area.

The compositor will reply with a configure event telling the expected new surface size. The operation is completed on the next buffer attach to this surface.

A maximized surface typically fills the entire output it is bound to, except for desktop element such as panels. This is the main difference between a maximized shell surface and a fullscreen shell surface.

The details depend on the compositor implementation.

wl_shell_surface::set_maximized arguments
output
Type: object

A.13.1.9. wl_shell_surface::set_title - set surface title

Set a short title for the surface.

This string may be used to identify the surface in a task bar, window list, or other user interface elements provided by the compositor.

The string must be encoded in UTF-8.

wl_shell_surface::set_title arguments
title
Type: string

A.13.1.10. wl_shell_surface::set_class - set surface class

Set a class for the surface.

The surface class identifies the general class of applications to which the surface belongs. A common convention is to use the file name (or the full path if it is a non-standard location) of the application's .desktop file as the class.

wl_shell_surface::set_class arguments
class_
Type: string

A.13.2. Events provided by wl_shell_surface

A.13.2.1. wl_shell_surface::ping - ping client

Ping a client to check if it is receiving events and sending requests. A client is expected to reply with a pong request.

wl_shell_surface::ping arguments
serial
Type: uint

A.13.2.2. wl_shell_surface::configure - suggest resize

The configure event asks the client to resize its surface.

The size is a hint, in the sense that the client is free to ignore it if it doesn't resize, pick a smaller size (to satisfy aspect ratio or resize in steps of NxM pixels).

The edges parameter provides a hint about how the surface was resized. The client may use this information to decide how to adjust its content to the new size (e.g. a scrolling area might adjust its content position to leave the viewable content unmoved).

The client is free to dismiss all but the last configure event it received.

The width and height arguments specify the size of the window in surface local coordinates.

wl_shell_surface::configure arguments
edges
Type: uint

width
Type: int

height
Type: int

A.13.2.3. wl_shell_surface::popup_done - popup interaction is done

The popup_done event is sent out when a popup grab is broken, that is, when the user clicks a surface that doesn't belong to the client owning the popup surface.

A.13.3. Enums provided by wl_shell_surface

A.13.3.1. wl_shell_surface::resize - edge values for resizing

These values are used to indicate which edge of a surface is being dragged in a resize operation. The server may use this information to adapt its behavior, e.g. choose an appropriate cursor image.

wl_shell_surface::resize values
none
Value: 0

top
Value: 1

bottom
Value: 2

left
Value: 4

top_left
Value: 5

bottom_left
Value: 6

right
Value: 8

top_right
Value: 9

bottom_right
Value: 10

A.13.3.2. wl_shell_surface::transient - details of transient behaviour

These flags specify details of the expected behaviour of transient surfaces. Used in the set_transient request.

wl_shell_surface::transient values

inactive
Value: 0x1

do not set keyboard focus

A.13.3.3. wl_shell_surface::fullscreen_method - different method to set the surface fullscreen

Hints to indicate to the compositor how to deal with a conflict between the dimensions of the surface and the dimensions of the output. The compositor is free to ignore this parameter.

wl_shell_surface::fullscreen_method values

default
Value: 0

no preference, apply default policy

scale
Value: 1

scale, preserve the surface's aspect ratio and center on output

driver
Value: 2

switch output mode to the smallest mode that can fit the surface, add black borders to compensate size mismatch

fill

Value: 3

no upscaling, center on output and add black borders to compensate size mismatch

A.14. wl_surface - an onscreen surface

A surface is a rectangular area that is displayed on the screen. It has a location, size and pixel contents.

The size of a surface (and relative positions on it) is described in surface local coordinates, which may differ from the buffer local coordinates of the pixel content, in case a buffer_transform or a buffer_scale is used.

Surfaces are also used for some special purposes, e.g. as cursor images for pointers, drag icons, etc.

A.14.1. Requests provided by wl_surface

A.14.1.1. wl_surface::destroy - delete surface

Deletes the surface and invalidates its object ID.

A.14.1.2. wl_surface::attach - set the surface contents

Set a buffer as the content of this surface.

The new size of the surface is calculated based on the buffer size transformed by the inverse buffer_transform and the inverse buffer_scale. This means that the supplied buffer must be an integer multiple of the buffer_scale.

The x and y arguments specify the location of the new pending buffer's upper left corner, relative to the current buffer's upper left corner, in surface local coordinates. In other words, the x and y, combined with the new surface size define in which directions the surface's size changes.

Surface contents are double-buffered state, see wl_surface.commit.

The initial surface contents are void; there is no content. wl_surface.attach assigns the given wl_buffer as the pending wl_buffer. wl_surface.commit makes the pending wl_buffer the new surface contents, and the size of the surface becomes the size calculated from the wl_buffer, as described above. After commit, there is no pending buffer until the next attach.

Committing a pending wl_buffer allows the compositor to read the pixels in the wl_buffer. The compositor may access the pixels at any time after the wl_surface.commit request. When the compositor will not access the pixels anymore, it will send the wl_buffer.release event. Only after receiving wl_buffer.release, the client may re-use the wl_buffer. A wl_buffer that has been attached and then replaced by another attach instead of committed will not receive a release event, and is not used by the compositor.

Destroying the wl_buffer after wl_buffer.release does not change the surface contents. However, if the client destroys the wl_buffer before receiving the wl_buffer.release event, the surface contents become undefined immediately.

If wl_surface.attach is sent with a NULL wl_buffer, the following wl_surface.commit will remove the surface content.

wl_surface::attach arguments

buffer
: Type: object

x
: Type: int

y
: Type: int

A.14.1.3. wl_surface::damage - mark part of the surface damaged

This request is used to describe the regions where the pending buffer is different from the current surface contents, and where the surface therefore needs to be repainted. The pending buffer must be set by wl_surface.attach before sending damage. The compositor ignores the parts of the damage that fall outside of the surface.

Damage is double-buffered state, see wl_surface.commit.

The damage rectangle is specified in surface local coordinates.

The initial value for pending damage is empty: no damage. wl_surface.damage adds pending damage: the new pending damage is the union of old pending damage and the given rectangle.

wl_surface.commit assigns pending damage as the current damage, and clears pending damage. The server will clear the current damage as it repaints the surface.

wl_surface::damage arguments

x
: Type: int

y
: Type: int

width
: Type: int

height
: Type: int

A.14.1.4. wl_surface::frame - request repaint feedback

Request notification when the next frame is displayed. Useful for throttling redrawing operations, and driving animations. The frame request will take effect on the next wl_surface.commit. The notification will only be posted for one frame unless requested again.

A server should avoid signalling the frame callbacks if the surface is not visible in any way, e.g. the surface is off-screen, or completely obscured by other opaque surfaces.

A client can request a frame callback even without an attach, damage, or any other state changes. wl_surface.commit triggers a display update, so the callback event will arrive after the next output refresh where the surface is visible.

The object returned by this request will be destroyed by the compositor after the callback is fired and as such the client must not attempt to use it after that point.

wl_surface::frame arguments
callback
Type: new_id

A.14.1.5. wl_surface::set_opaque_region - set opaque region

This request sets the region of the surface that contains opaque content.

The opaque region is an optimization hint for the compositor that lets it optimize out redrawing of content behind opaque regions. Setting an opaque region is not required for correct behaviour, but marking transparent content as opaque will result in repaint artifacts.

The opaque region is specified in surface local coordinates.

The compositor ignores the parts of the opaque region that fall outside of the surface.

Opaque region is double-buffered state, see wl_surface.commit.

wl_surface.set_opaque_region changes the pending opaque region. wl_surface.commit copies the pending region to the current region. Otherwise, the pending and current regions are never changed.

The initial value for opaque region is empty. Setting the pending opaque region has copy semantics, and the wl_region object can be destroyed immediately. A NULL wl_region causes the pending opaque region to be set to empty.

wl_surface::set_opaque_region arguments
region
Type: object

A.14.1.6. wl_surface::set_input_region - set input region

This request sets the region of the surface that can receive pointer and touch events.

Input events happening outside of this region will try the next surface in the server surface stack. The compositor ignores the parts of the input region that fall outside of the surface.

The input region is specified in surface local coordinates.

Input region is double-buffered state, see wl_surface.commit.

wl_surface.set_input_region changes the pending input region. wl_surface.commit copies the pending region to the current region. Otherwise the pending and current regions are never changed, except cursor and icon surfaces are special cases, see wl_pointer.set_cursor and wl_data_device.start_drag.

The initial value for input region is infinite. That means the whole surface will accept input. Setting the pending input region has copy semantics, and the wl_region object can be destroyed immediately. A NULL wl_region causes the input region to be set to infinite.

wl_surface::set_input_region arguments
region
Type: object

A.14.1.7. wl_surface::commit - commit pending surface state

Surface state (input, opaque, and damage regions, attached buffers, etc.) is double-buffered. Protocol requests modify the pending state, as opposed to current state in use by the compositor. Commit

request atomically applies all pending state, replacing the current state. After commit, the new pending state is as documented for each related request.

On commit, a pending wl_buffer is applied first, all other state second. This means that all coordinates in double-buffered state are relative to the new wl_buffer coming into use, except for wl_surface.attach itself. If there is no pending wl_buffer, the coordinates are relative to the current surface contents.

All requests that need a commit to become effective are documented to affect double-buffered state.

Other interfaces may add further double-buffered surface state.

A.14.1.8. wl_surface::set_buffer_transform - sets the buffer transformation

This request sets an optional transformation on how the compositor interprets the contents of the buffer attached to the surface. The accepted values for the transform parameter are the values for wl_output.transform.

Buffer transform is double-buffered state, see wl_surface.commit.

A newly created surface has its buffer transformation set to normal.

The purpose of this request is to allow clients to render content according to the output transform, thus permiting the compositor to use certain optimizations even if the display is rotated. Using hardware overlays and scanning out a client buffer for fullscreen surfaces are examples of such optimizations. Those optimizations are highly dependent on the compositor implementation, so the use of this request should be considered on a case-by-case basis.

Note that if the transform value includes 90 or 270 degree rotation, the width of the buffer will become the surface height and the height of the buffer will become the surface width.

wl_surface::set_buffer_transform arguments

transform

Type: int

A.14.1.9. wl_surface::set_buffer_scale - sets the buffer scaling factor

This request sets an optional scaling factor on how the compositor interprets the contents of the buffer attached to the window.

Buffer scale is double-buffered state, see wl_surface.commit.

A newly created surface has its buffer scale set to 1.

The purpose of this request is to allow clients to supply higher resolution buffer data for use on high resolution outputs. Its intended that you pick the same buffer scale as the scale of the output that the surface is displayed on.This means the compositor can avoid scaling when rendering the surface on that output.

Note that if the scale is larger than 1, then you have to attach a buffer that is larger (by a factor of scale in each dimension) than the desired surface size.

wl_surface::set_buffer_scale arguments

scale

Type: int

A.14.2. Events provided by wl_surface

A.14.2.1. wl_surface::enter - surface enters an output

This is emitted whenever a surface's creation, movement, or resizing results in some part of it being within the scanout region of an output.

Note that a surface may be overlapping with zero or more outputs.

wl_surface::enter arguments

output

Type: object

A.14.2.2. wl_surface::leave - surface leaves an output

This is emitted whenever a surface's creation, movement, or resizing results in it no longer having any part of it within the scanout region of an output.

wl_surface::leave arguments

output

Type: object

A.15. wl_seat - group of input devices

A seat is a group of keyboards, pointer and touch devices. This object is published as a global during start up, or when such a device is hot plugged. A seat typically has a pointer and maintains a keyboard focus and a pointer focus.

A.15.1. Requests provided by wl_seat

A.15.1.1. wl_seat::get_pointer - return pointer object

The ID provided will be initialized to the wl_pointer interface for this seat.

This request only takes effect if the seat has the pointer capability.

wl_seat::get_pointer arguments

id

Type: new_id

A.15.1.2. wl_seat::get_keyboard - return keyboard object

The ID provided will be initialized to the wl_keyboard interface for this seat.

This request only takes effect if the seat has the keyboard capability.

wl_seat::get_keyboard arguments

id

Type: new_id

A.15.1.3. wl_seat::get_touch - return touch object

The ID provided will be initialized to the wl_touch interface for this seat.

This request only takes effect if the seat has the touch capability.

wl_seat::get_touch arguments

id

Type: new_id

A.15.2. Events provided by wl_seat

A.15.2.1. wl_seat::capabilities - seat capabilities changed

This is emitted whenever a seat gains or loses the pointer, keyboard or touch capabilities. The argument is a capability enum containing the complete set of capabilities this seat has.

wl_seat::capabilities arguments

capabilities

Type: uint

A.15.2.2. wl_seat::name - unique identifier for this seat

In a multiseat configuration this can be used by the client to help identify which physical devices the seat represents. Based on the seat configuration used by the compositor.

wl_seat::name arguments

name

Type: string

A.15.3. Enums provided by wl_seat

A.15.3.1. wl_seat::capability - seat capability bitmask

This is a bitmask of capabilities this seat has; if a member is set, then it is present on the seat.

wl_seat::capability values

pointer

Value: 1

The seat has pointer devices

keyboard

Value: 2

The seat has one or more keyboards

touch

Value: 4

The seat has touch devices

A.16. wl_pointer - pointer input device

The wl_pointer interface represents one or more input devices, such as mice, which control the pointer location and pointer_focus of a seat.

The wl_pointer interface generates motion, enter and leave events for the surfaces that the pointer is located over, and button and axis events for button presses, button releases and scrolling.

A.16.1. Requests provided by wl_pointer

A.16.1.1. wl_pointer::set_cursor - set the pointer surface

Set the pointer surface, i.e., the surface that contains the pointer image (cursor). This request only takes effect if the pointer focus for this device is one of the requesting client's surfaces or the surface parameter is the current pointer surface. If there was a previous surface set with this request it is replaced. If surface is NULL, the pointer image is hidden.

The parameters hotspot_x and hotspot_y define the position of the pointer surface relative to the pointer location. Its top-left corner is always at (x, y) - (hotspot_x, hotspot_y), where (x, y) are the coordinates of the pointer location, in surface local coordinates.

On surface.attach requests to the pointer surface, hotspot_x and hotspot_y are decremented by the x and y parameters passed to the request. Attach must be confirmed by wl_surface.commit as usual.

The hotspot can also be updated by passing the currently set pointer surface to this request with new values for hotspot_x and hotspot_y.

The current and pending input regions of the wl_surface are cleared, and wl_surface.set_input_region is ignored until the wl_surface is no longer used as the cursor. When the use as a cursor ends, the current and pending input regions become undefined, and the wl_surface is unmapped.

wl_pointer::set_cursor arguments

serial
: Type: uint

  serial of the enter event

surface
: Type: object

hotspot_x
: Type: int

  x coordinate in surface-relative coordinates

hotspot_y
: Type: int

  y coordinate in surface-relative coordinates

A.16.1.2. wl_pointer::release - release the pointer object

A.16.2. Events provided by wl_pointer

A.16.2.1. wl_pointer::enter - enter event

Notification that this seat's pointer is focused on a certain surface.

When an seat's focus enters a surface, the pointer image is undefined and a client should respond to this event by setting an appropriate pointer image with the set_cursor request.

wl_pointer::enter arguments

serial
: Type: uint

surface
: Type: object

surface_x
: Type: fixed

  x coordinate in surface-relative coordinates

surface_y
: Type: fixed

  y coordinate in surface-relative coordinates

A.16.2.2. wl_pointer::leave - leave event

Notification that this seat's pointer is no longer focused on a certain surface.

The leave notification is sent before the enter notification for the new focus.

wl_pointer::leave arguments

serial
: Type: uint

surface
: Type: object

A.16.2.3. wl_pointer::motion - pointer motion event

Notification of pointer location change. The arguments surface_x and surface_y are the location relative to the focused surface.

wl_pointer::motion arguments

time
: Type: uint

  timestamp with millisecond granularity

surface_x
: Type: fixed

  x coordinate in surface-relative coordinates

surface_y
: Type: fixed

  y coordinate in surface-relative coordinates

A.16.2.4. wl_pointer::button - pointer button event

Mouse button click and release notifications.

The location of the click is given by the last motion or enter event. The time argument is a timestamp with millisecond granularity, with an undefined base.

wl_pointer::button arguments

serial
: Type: uint

time
: Type: uint

  timestamp with millisecond granularity

button
: Type: uint

state
: Type: uint

A.16.2.5. wl_pointer::axis - axis event

Scroll and other axis notifications.

For scroll events (vertical and horizontal scroll axes), the value parameter is the length of a vector along the specified axis in a coordinate space identical to those of motion events, representing a relative movement along the specified axis.

For devices that support movements non-parallel to axes multiple axis events will be emitted.

When applicable, for example for touch pads, the server can choose to emit scroll events where the motion vector is equivalent to a motion event vector.

When applicable, clients can transform its view relative to the scroll distance.

wl_pointer::axis arguments

time
: Type: uint

  timestamp with millisecond granularity

axis
: Type: uint

value
: Type: fixed

A.16.3. Enums provided by wl_pointer

A.16.3.1. wl_pointer::button_state - physical button state

Describes the physical state of a button which provoked the button event.

wl_pointer::button_state values

released
: Value: 0

  The button is not pressed

pressed
: Value: 1

  The button is pressed

A.16.3.2. wl_pointer::axis - axis types

Describes the axis types of scroll events.

wl_pointer::axis values

vertical_scroll
: Value: 0

horizontal_scroll
: Value: 1

A.17. wl_keyboard - keyboard input device

The wl_keyboard interface represents one or more keyboards associated with a seat.

A.17.1. Requests provided by wl_keyboard

A.17.1.1. wl_keyboard::release - release the keyboard object

A.17.2. Events provided by wl_keyboard

A.17.2.1. wl_keyboard::keymap - keyboard mapping

This event provides a file descriptor to the client which can be memory-mapped to provide a keyboard mapping description.

wl_keyboard::keymap arguments

format
: Type: uint

fd
: Type: fd

size
: Type: uint

A.17.2.2. wl_keyboard::enter - enter event

Notification that this seat's keyboard focus is on a certain surface.

wl_keyboard::enter arguments

serial
: Type: uint

surface
: Type: object

keys
: Type: array

  the currently pressed keys

A.17.2.3. wl_keyboard::leave - leave event

Notification that this seat's keyboard focus is no longer on a certain surface.

The leave notification is sent before the enter notification for the new focus.

wl_keyboard::leave arguments

serial

Type: uint

surface

Type: object

A.17.2.4. wl_keyboard::key - key event

A key was pressed or released. The time argument is a timestamp with millisecond granularity, with an undefined base.

wl_keyboard::key arguments

serial

Type: uint

time

Type: uint

timestamp with millisecond granularity

key

Type: uint

state

Type: uint

A.17.2.5. wl_keyboard::modifiers - modifier and group state

Notifies clients that the modifier and/or group state has changed, and it should update its local state.

wl_keyboard::modifiers arguments

serial

Type: uint

mods_depressed

Type: uint

mods_latched

Type: uint

mods_locked

Type: uint

group

Type: uint

A.17.3. Enums provided by wl_keyboard

A.17.3.1. wl_keyboard::keymap_format - keyboard mapping format

This specifies the format of the keymap provided to the client with the wl_keyboard.keymap event.

wl_keyboard::keymap_format values

no_keymap
: Value: 0

  no keymap; client must understand how to interpret the raw keycode

xkb_v1
: Value: 1

  libxkbcommon compatible

A.17.3.2. wl_keyboard::key_state - physical key state

Describes the physical state of a key which provoked the key event.

wl_keyboard::key_state values

released
: Value: 0

  key is not pressed

pressed
: Value: 1

  key is pressed

A.18. wl_touch - touchscreen input device

The wl_touch interface represents a touchscreen associated with a seat.

Touch interactions can consist of one or more contacts. For each contact, a series of events is generated, starting with a down event, followed by zero or more motion events, and ending with an up event. Events relating to the same contact point can be identified by the ID of the sequence.

A.18.1. Requests provided by wl_touch

A.18.1.1. wl_touch::release - release the touch object

A.18.2. Events provided by wl_touch

A.18.2.1. wl_touch::down - touch down event and beginning of a touch sequence

A new touch point has appeared on the surface. This touch point is assigned a unique @id. Future events from this touchpoint reference this ID. The ID ceases to be valid after a touch up event and may be re-used in the future.

wl_touch::down arguments

serial
: Type: uint

time
: Type: uint

  timestamp with millisecond granularity

surface
: Type: object

id
: Type: int

  the unique ID of this touch point

x
: Type: fixed

  x coordinate in surface-relative coordinates

y
: Type: fixed

  y coordinate in surface-relative coordinates

A.18.2.2. wl_touch::up - end of a touch event sequence

The touch point has disappeared. No further events will be sent for this touchpoint and the touch point's ID is released and may be re-used in a future touch down event.

wl_touch::up arguments

serial
: Type: uint

time
: Type: uint

  timestamp with millisecond granularity

id
: Type: int

  the unique ID of this touch point

A.18.2.3. wl_touch::motion - update of touch point coordinates

A touchpoint has changed coordinates.

wl_touch::motion arguments

time
: Type: uint

  timestamp with millisecond granularity

id
: Type: int

  the unique ID of this touch point

x
: Type: fixed

  x coordinate in surface-relative coordinates

y
: Type: fixed

  y coordinate in surface-relative coordinates

A.18.2.4. wl_touch::frame - end of touch frame event

Indicates the end of a contact point list.

A.18.2.5. wl_touch::cancel - touch session cancelled

Sent if the compositor decides the touch stream is a global gesture. No further events are sent to the clients from that particular gesture. Touch cancellation applies to all touch points currently active on this client's surface. The client is responsible for finalizing the touch points, future touch points on this surface may re-use the touch point ID.

A.19. wl_output - compositor output region

An output describes part of the compositor geometry. The compositor works in the 'compositor coordinate system' and an output corresponds to rectangular area in that space that is actually visible. This typically corresponds to a monitor that displays part of the compositor space. This object is published as global during start up, or when a monitor is hotplugged.

A.19.1. Events provided by wl_output

A.19.1.1. wl_output::geometry - properties of the output

The geometry event describes geometric properties of the output. The event is sent when binding to the output object and whenever any of the properties change.

wl_output::geometry arguments

x
: Type: int

  x position within the global compositor space

y
: Type: int

  y position within the global compositor space

physical_width
: Type: int

  width in millimeters of the output

physical_height
: Type: int

  height in millimeters of the output

subpixel
: Type: int

  subpixel orientation of the output

make
: Type: string

  textual description of the manufacturer

model
: Type: string

  textual description of the model

transform
: Type: int

  transform that maps framebuffer to output

A.19.1.2. wl_output::mode - advertise available modes for the output

The mode event describes an available mode for the output.

The event is sent when binding to the output object and there will always be one mode, the current mode. The event is sent again if an output changes mode, for the mode that is now current. In other words, the current mode is always the last mode that was received with the current flag set.

The size of a mode is given in physical hardware units of the output device. This is not necessarily the same as the output size in the global compositor space. For instance, the output may be scaled, as described in wl_output.scale, or transformed , as described in wl_output.transform.

wl_output::mode arguments

flags
: Type: uint

  bitfield of mode flags

width
: Type: int

  width of the mode in hardware units

height
: Type: int

  height of the mode in hardware units

refresh
: Type: int

  vertical refresh rate in mHz

A.19.1.3. wl_output::done - sent all information about output

This event is sent after all other properties has been sent after binding to the output object and after any other property changes done after that. This allows changes to the output properties to be seen as atomic, even if they happen via multiple events.

A.19.1.4. wl_output::scale - output scaling properties

This event contains scaling geometry information that is not in the geometry event. It may be sent after binding the output object or if the output scale changes later. If it is not sent, the client should assume a scale of 1.

A scale larger than 1 means that the compositor will automatically scale surface buffers by this amount when rendering. This is used for very high resolution displays where applications rendering at the native resolution would be too small to be legible.

It is intended that scaling aware clients track the current output of a surface, and if it is on a scaled output it should use wl_surface.set_buffer_scale with the scale of the output. That way the compositor can avoid scaling the surface, and the client can supply a higher detail image.

wl_output::scale arguments

factor

Type: int

scaling factor of output

A.19.2. Enums provided by wl_output

A.19.2.1. wl_output::subpixel - subpixel geometry information

This enumeration describes how the physical pixels on an output are layed out.

wl_output::subpixel values

unknown

Value: 0

none

Value: 1

horizontal_rgb

Value: 2

horizontal_bgr

Value: 3

vertical_rgb

Value: 4

vertical_bgr

Value: 5

A.19.2.2. wl_output::transform - transform from framebuffer to output

This describes the transform that a compositor will apply to a surface to compensate for the rotation or mirroring of an output device.

The flipped values correspond to an initial flip around a vertical axis followed by rotation.

The purpose is mainly to allow clients render accordingly and tell the compositor, so that for fullscreen surfaces, the compositor will still be able to scan out directly from client surfaces.

wl_output::transform values

normal
: Value: 0

90
: Value: 1

180
: Value: 2

270
: Value: 3

flipped
: Value: 4

flipped_90
: Value: 5

flipped_180
: Value: 6

flipped_270
: Value: 7

A.19.2.3. wl_output::mode - mode information

These flags describe properties of an output mode. They are used in the flags bitfield of the mode event.

wl_output::mode values

current
: Value: 0x1

  indicates this is the current mode

preferred
: Value: 0x2

  indicates this is the preferred mode

A.20. wl_region - region interface

A region object describes an area.

Region objects are used to describe the opaque and input regions of a surface.

A.20.1. Requests provided by wl_region

A.20.1.1. wl_region::destroy - destroy region

Destroy the region. This will invalidate the object ID.

A.20.1.2. wl_region::add - add rectangle to region

Add the specified rectangle to the region.

wl_region::add arguments

x
: Type: int

y
: Type: int

width
: Type: int

height
: Type: int

A.20.1.3. wl_region::subtract - subtract rectangle from region

Subtract the specified rectangle from the region.

wl_region::subtract arguments

x
: Type: int

y
: Type: int

width
: Type: int

height
: Type: int

Wayland Library

The open-source reference implementation of Wayland protocol is split in two C libraries, *libwayland-server* and *libwayland-client*. Their main responsibility is to handle the Inter-process communication (*IPC*) with each other, therefore guaranteeing the protocol objects marshaling and messages synchronization.

This Chapter describes in detail each library's methods and their helpers, aiming implementors who can use for building Wayland clients and servers; respectively at *Section 5.1, “Client API”* and *Section 5.2, “Server API”*.

5.1. Client API

Following is the Wayland library classes for the Client (*libwayland-client*). Note that most of the procedures are related with IPC, which is the main responsibility of the library.

wl_display - Represents a connection to the compositor and acts as a proxy to the wl_display singleton object.

A wl_display object represents a client connection to a Wayland compositor. It is created with either wl_display_connect() or wl_display_connect_to_fd(). A connection is terminated using wl_display_disconnect().

A wl_display is also used as the wl_proxy for the wl_display singleton object on the compositor side.

A wl_display object handles all the data sent from and to the compositor. When a wl_proxy marshals a request, it will write its wire representation to the display's write buffer. The data is sent to the compositor when the client calls wl_display_flush().

Incoming data is handled in two steps: queueing and dispatching. In the queue step, the data coming from the display fd is interpreted and added to a queue. On the dispatch step, the handler for the incoming event set by the client on the corresponding wl_proxy is called.

A wl_display has at least one event queue, called the main queue. Clients can create additional event queues with wl_display_create_queue() and assign wl_proxy's to it. Events occurring in a particular proxy are always queued in its assigned queue. A client can ensure that a certain assumption, such as holding a lock or running from a given thread, is true when a proxy event handler is called by assigning that proxy to an event queue and making sure that this queue is only dispatched when the assumption holds.

The main queue is dispatched by calling wl_display_dispatch(). This will dispatch any events queued on the main queue and attempt to read from the display fd if its empty. Events read are then queued on the appropriate queues according to the proxy assignment. Calling that function makes the calling thread the main thread.

A user created queue is dispatched with wl_display_dispatch_queue(). If there are no events to dispatch this function will block. If this is called by the main thread, this will attempt to read data from the display fd and queue any events on the appropriate queues. If calling from any other thread, the function will block until the main thread queues an event on the queue being dispatched.

A real world example of event queue usage is Mesa's implementation of eglSwapBuffers() for the Wayland platform. This function might need to block until a frame callback is received, but dispatching the main queue could cause an event handler on the client to start drawing again. This

problem is solved using another event queue, so that only the events handled by the EGL code are dispatched during the block.

This creates a problem where the main thread dispatches a non-main queue, reading all the data from the display fd. If the application would call poll(2) after that it would block, even though there might be events queued on the main queue. Those events should be dispatched with wl_display_dispatch_pending() before flushing and blocking.

wl_event_queue - A queue for wl_proxy object events.

Event queues allows the events on a display to be handled in a thread-safe manner. See wl_display for details.

wl_list - doubly-linked list

The list head is of "struct wl_list" type, and must be initialized using wl_list_init(). All entries in the list must be of the same type. The item type must have a "struct wl_list" member. This member will be initialized by wl_list_insert(). There is no need to call wl_list_init() on the individual item. To query if the list is empty in O(1), use wl_list_empty().

Let's call the list reference "struct wl_list foo_list", the item type as "item_t", and the item member as "struct wl_list link".

The following code will initialize a list: struct wl_list foo_list; struct item_t { int foo; struct wl_list link; }; struct item_t item1, item2, item3; wl_list_init(&foo_list); wl_list_insert(&foo_list, &item1.link); Pushes item1 at the head wl_list_insert(&foo_list, &item2.link); Pushes item2 at the head wl_list_insert(&item2.link, &item3.link); Pushes item3 after item2

The list now looks like [item2, item3, item1]

Will iterate the list in ascending order: item_t *item; wl_list_for_each(item, foo_list, link) { Do_something_with_item(item);

wl_proxy - Represents a protocol object on the client side.

A wl_proxy acts as a client side proxy to an object existing in the compositor. The proxy is responsible for converting requests made by the clients with wl_proxy_marshal() into Wayland's wire format. Events coming from the compositor are also handled by the proxy, which will in turn call the handler set with wl_proxy_add_listener().

With the exception of function wl_proxy_set_queue(), functions accessing a wl_proxy are not normally used by client code. Clients should normally use the higher level interface generated by the scanner to interact with compositor objects.

Methods for the respective classes.

wl_display_create_queue - Create a new event queue for this display.

```
struct wl_event_queue * wl_display_create_queue(struct wl_display *display)
```

display

The display context object

Returns:

A new event queue associated with this display or NULL on failure.

wl_display_connect_to_fd - Connect to Wayland display on an already open fd.

```
struct wl_display * wl_display_connect_to_fd(int fd)
```

fd
: The fd to use for the connection

Returns:
: A wl_display object or NULL on failure

The *wl_display* takes ownership of the fd and will close it when the display is destroyed. The fd will also be closed in case of failure.

wl_display_connect - Connect to a Wayland display.

```
struct wl_display * wl_display_connect(const char *name)
```

name
: Name of the Wayland display to connect to

Returns:
: A wl_display object or NULL on failure

Connect to the Wayland display named name. If name is NULL, its value will be replaced with the WAYLAND_DISPLAY environment variable if it is set, otherwise display "wayland-0" will be used.

wl_display_disconnect - Close a connection to a Wayland display.

```
void wl_display_disconnect(struct wl_display *display)
```

display
: The display context object

Close the connection to display and free all resources associated with it.

wl_display_get_fd - Get a display context's file descriptor.

```
int wl_display_get_fd(struct wl_display *display)
```

display
: The display context object

Returns:
: Display object file descriptor

Return the file descriptor associated with a display so it can be integrated into the client's main loop.

wl_display_roundtrip - Block until all pending request are processed by the server.

```
int wl_display_roundtrip(struct wl_display *display)
```

display
: The display context object

Returns:
: The number of dispatched events on success or -1 on failure

Blocks until the server process all currently issued requests and sends out pending events on all event queues.

wl_display_read_events - Read events from display file descriptor.

```
int wl_display_read_events(struct wl_display *display)
```

display
The display context object

Returns:
0 on success or -1 on error. In case of error errno will be set accordingly

This will read events from the file descriptor for the display. This function does not dispatch events, it only reads and queues events into their corresponding event queues. If no data is avilable on the file descriptor, *wl_display_read_events()* returns immediately. To dispatch events that may have been queued, call *wl_display_dispatch_pending()* or *wl_display_dispatch_queue_pending()*.

Before calling this function, *wl_display_prepare_read()* must be called first.

wl_display_prepare_read - Prepare to read events after polling file descriptor.

```
int wl_display_prepare_read(struct wl_display *display)
```

display
The display context object

Returns:
0 on success or -1 if event queue was not empty

This function must be called before reading from the file descriptor using *wl_display_read_events()*. Calling *wl_display_prepare_read()* announces the calling threads intention to read and ensures that until the thread is ready to read and calls *wl_display_read_events()*, no other thread will read from the file descriptor. This only succeeds if the event queue is empty though, and if there are undispatched events in the queue, -1 is returned and errno set to EAGAIN.

If a thread successfully calls *wl_display_prepare_read()*, it must either call *wl_display_read_events()* when it's ready or cancel the read intention by calling *wl_display_cancel_read()*.

Use this function before polling on the display fd or to integrate the fd into a toolkit event loop in a race-free way. Typically, a toolkit will call *wl_display_dispatch_pending()* before sleeping, to make sure it doesn't block with unhandled events. Upon waking up, it will assume the file descriptor is readable and read events from the fd by calling *wl_display_dispatch()*. Simplified, we have:

wl_display_dispatch_pending(display); wl_display_flush(display); poll(fds, nfds, -1); wl_display_dispatch(display);

There are two races here: first, before blocking in poll(), the fd could become readable and another thread reads the events. Some of these events may be for the main queue and the other thread will queue them there and then the main thread will go to sleep in poll(). This will stall the application, which could be waiting for a event to kick of the next animation frame, for example.

The other race is immediately after poll(), where another thread could preempt and read events before the main thread calls *wl_display_dispatch()*. This call now blocks and starves the other fds in the event loop.

A correct sequence would be:

while (wl_display_prepare_read(display) != 0) wl_display_dispatch_pending(display); wl_display_flush(display); poll(fds, nfds, -1); wl_display_read_events(display); wl_display_dispatch_pending(display);

Here we call *wl_display_prepare_read()*, which ensures that between returning from that call and eventually calling *wl_display_read_events()*, no other thread will read from the fd and queue events in our queue. If the call to *wl_display_prepare_read()* fails, we dispatch the pending events and try again until we're successful.

wl_display_cancel_read - Release exclusive access to display file descriptor.

```
void wl_display_cancel_read(struct wl_display *display)
```

display
: The display context object

This releases the exclusive access. Useful for canceling the lock when a timed out poll returns fd not readable and we're not going to read from the fd anytime soon.

wl_display_dispatch_queue - Dispatch events in an event queue.

```
int wl_display_dispatch_queue(struct wl_display *display, struct wl_event_queue *queue)
```

display
: The display context object

queue
: The event queue to dispatch

Returns:
: The number of dispatched events on success or -1 on failure

Dispatch all incoming events for objects assigned to the given event queue. On failure -1 is returned and errno set appropriately.

This function blocks if there are no events to dispatch. If calling from the main thread, it will block reading data from the display fd. For other threads this will block until the main thread queues events on the queue passed as argument.

wl_display_dispatch_queue_pending - Dispatch pending events in an event queue.

```
int wl_display_dispatch_queue_pending(struct wl_display *display, struct wl_event_queue
 *queue)
```

display
: The display context object

queue
: The event queue to dispatch

Returns:
: The number of dispatched events on success or -1 on failure

Dispatch all incoming events for objects assigned to the given event queue. On failure -1 is returned and errno set appropriately. If there are no events queued, this function returns immediately.

- Since: 1.0.2

wl_display_dispatch - Process incoming events.

```
int wl_display_dispatch(struct wl_display *display)
```

display
: The display context object

Returns:
: The number of dispatched events on success or -1 on failure

Dispatch the display's main event queue.

If the main event queue is empty, this function blocks until there are events to be read from the display fd. Events are read and queued on the appropriate event queues. Finally, events on the main event queue are dispatched.

Note: It is not possible to check if there are events on the main queue or not. For dispatching main queue events without blocking, see wl_display_dispatch_pending().Calling this will release the display file descriptor if this thread acquired it using wl_display_acquire_fd().

- See also: *wl_display_dispatch_pending() wl_display_dispatch_queue()*

wl_display_dispatch_pending - Dispatch main queue events without reading from the display fd.

```
int wl_display_dispatch_pending(struct wl_display *display)
```

display
: The display context object

Returns:
: The number of dispatched events or -1 on failure

This function dispatches events on the main event queue. It does not attempt to read the display fd and simply returns zero if the main queue is empty, i.e., it doesn't block.

This is necessary when a client's main loop wakes up on some fd other than the display fd (network socket, timer fd, etc) and calls *wl_display_dispatch_queue()* from that callback. This may queue up events in the main queue while reading all data from the display fd. When the main thread returns to the main loop to block, the display fd no longer has data, causing a call to poll(2) (or similar functions) to block indefinitely, even though there are events ready to dispatch.

To proper integrate the wayland display fd into a main loop, the client should always call *wl_display_dispatch_pending()* and then *wl_display_flush()* prior to going back to sleep. At that point, the fd typically doesn't have data so attempting I/O could block, but events queued up on the main queue should be dispatched.

A real-world example is a main loop that wakes up on a timerfd (or a sound card fd becoming writable, for example in a video player), which then triggers GL rendering and eventually eglSwapBuffers(). eglSwapBuffers() may call *wl_display_dispatch_queue()* if it didn't receive the frame event for the previous frame, and as such queue events in the main queue.

Note: Calling this makes the current thread the main one.

- See also: *wl_display_dispatch() wl_display_dispatch_queue() wl_display_flush()*

wl_display_get_error - Retrieve the last error that occurred on a display.

```
int wl_display_get_error(struct wl_display *display)
```

display
: The display context object

Returns:
: The last error that occurred on display or 0 if no error occurred

Return the last error that occurred on the display. This may be an error sent by the server or caused by the local client.

Note: Errors are fatal. If this function returns non-zero the display can no longer be used.

wl_display_flush - Send all buffered requests on the display to the server.

```
int wl_display_flush(struct wl_display *display)
```

display
: The display context object

Returns:
: The number of bytes sent on success or -1 on failure

Send all buffered data on the client side to the server. Clients should call this function before blocking. On success, the number of bytes sent to the server is returned. On failure, this function returns -1 and errno is set appropriately.

wl_display_flush() never blocks. It will write as much data as possible, but if all data could not be written, errno will be set to EAGAIN and -1 returned. In that case, use poll on the display file descriptor to wait for it to become writable again.

wl_event_queue_destroy - Destroy an event queue.

```
void wl_event_queue_destroy(struct wl_event_queue *queue)
```

queue
: The event queue to be destroyed

Destroy the given event queue. Any pending event on that queue is discarded.

The *wl_display* object used to create the queue should not be destroyed until all event queues created with it are destroyed with this function.

wl_proxy_create - Create a proxy object with a given interface.

```
struct wl_proxy * wl_proxy_create(struct wl_proxy *factory, const struct wl_interface
 *interface)
```

factory
: Factory proxy object

interface
: Interface the proxy object should use

Returns:
: A newly allocated proxy object or NULL on failure

This function creates a new proxy object with the supplied interface. The proxy object will have an id assigned from the client id space. The id should be created on the compositor side by sending an appropriate request with *wl_proxy_marshal()*.

The proxy will inherit the display and event queue of the factory object.

Note: This should not normally be used by non-generated code.

- See also: *wl_display wl_event_queue wl_proxy_marshal()*

wl_proxy_destroy - Destroy a proxy object.

```
void wl_proxy_destroy(struct wl_proxy *proxy)
```

proxy
: The proxy to be destroyed

wl_proxy_add_listener - Set a proxy's listener.

```
int wl_proxy_add_listener(struct wl_proxy *proxy, void(**implementation)(void), void
 *data)
```

proxy
: The proxy object

implementation
: The listener to be added to proxy

data
: User data to be associated with the proxy

Returns:
: 0 on success or -1 on failure

Set proxy's listener to implementation and its user data to data. If a listener has already been set, this function fails and nothing is changed.

implementation is a vector of function pointers. For an opcode n, implementation[n] should point to the handler of n for the given object.

wl_proxy_get_listener - Get a proxy's listener.

```
const void * wl_proxy_get_listener(struct wl_proxy *proxy)
```

proxy
: The proxy object

Returns:
: The address of the proxy's listener or NULL if no listener is set

Gets the address to the proxy's listener; which is the listener set with *wl_proxy_add_listener*.

This function is useful in client with multiple listeners on the same interface to allow the identification of which code to eexecute.

wl_proxy_add_dispatcher - Set a proxy's listener (with dispatcher)

```
int wl_proxy_add_dispatcher(struct wl_proxy *proxy, wl_dispatcher_func_t dispatcher,
 const void *implementation, void *data)
```

proxy
: The proxy object

dispatcher
: The dispatcher to be used for this proxy

implementation
: The dispatcher-specific listener implementation

data
: User data to be associated with the proxy

Returns:
: 0 on success or -1 on failure

Set proxy's listener to use dispatcher_func as its dispatcher and dispatcher_data as its dispatcher-specific implementation and its user data to data. If a listener has already been set, this function fails and nothing is changed.

The exact details of dispatcher_data depend on the dispatcher used. This function is intended to be used by language bindings, not user code.

wl_proxy_marshal - Prepare a request to be sent to the compositor.

```
void wl_proxy_marshal(struct wl_proxy *proxy, uint32_t opcode,...)
```

proxy
: The proxy object

opcode
: Opcode of the request to be sent

...
: Extra arguments for the given request

Translates the request given by opcode and the extra arguments into the wire format and write it to the connection buffer.

The example below creates a proxy object with the wl_surface_interface using a wl_compositor factory interface and sends the compositor.create_surface request using *wl_proxy_marshal()*. Note the id is the extra argument to the request as specified by the protocol.

```
*id=wl_proxy_create((structwl_proxy*)wl_compositor,
*&wl_surface_interface);
*wl_proxy_marshal((structwl_proxy*)wl_compositor,
*WL_COMPOSITOR_CREATE_SURFACE,id);
*
```

Note: This should not normally be used by non-generated code.

- See also: *wl_proxy_create()*

wl_proxy_marshal_array - Prepare a request to be sent to the compositor.

```
void wl_proxy_marshal_array(struct wl_proxy *proxy, uint32_t opcode, union wl_argument
 *args)
```

proxy
: The proxy object

opcode
: Opcode of the request to be sent

args
: Extra arguments for the given request

Translates the request given by opcode and the extra arguments into the wire format and write it to the connection buffer. This version takes an array of the union type *wl_argument*.

Note: This is intended to be used by language bindings and not in non-generated code.

- See also: *wl_proxy_marshal()*

wl_proxy_set_user_data - Set the user data associated with a proxy.

```
void wl_proxy_set_user_data(struct wl_proxy *proxy, void *user_data)
```

proxy
: The proxy object

user_data
: The data to be associated with proxy

Set the user data associated with proxy. When events for this proxy are received, user_data will be supplied to its listener.

wl_proxy_get_user_data - Get the user data associated with a proxy.

```
void * wl_proxy_get_user_data(struct wl_proxy *proxy)
```

proxy
: The proxy object

Returns:
: The user data associated with proxy

wl_proxy_get_id - Get the id of a proxy object.

```
uint32_t wl_proxy_get_id(struct wl_proxy *proxy)
```

proxy
: The proxy object

Returns:
: The id the object associated with the proxy

wl_proxy_get_class - Get the interface name (class) of a proxy object.

```
const char * wl_proxy_get_class(struct wl_proxy *proxy)
```

proxy
: The proxy object

Returns:
: The interface name of the object associated with the proxy

wl_proxy_set_queue - Assign a proxy to an event queue.

```
void wl_proxy_set_queue(struct wl_proxy *proxy, struct wl_event_queue *queue)
```

proxy
: The proxy object

queue
: The event queue that will handle this proxy

Assign proxy to event queue. Events coming from proxy will be queued in queue instead of the display's main queue.

- See also: *wl_display_dispatch_queue()*

wl_display_prepare_read_queue -

```
int wl_display_prepare_read_queue(struct wl_display *display, struct wl_event_queue
 *queue)
```

wl_log_set_handler_client -

```
void wl_log_set_handler_client(wl_log_func_t handler)
```

wl_list_init -

```
void wl_list_init(struct wl_list *list)
```

wl_list_insert -

```
void wl_list_insert(struct wl_list *list, struct wl_list *elm)
```

wl_list_remove -

```
void wl_list_remove(struct wl_list *elm)
```

wl_list_length -

```
int wl_list_length(const struct wl_list *list)
```

wl_list_empty -

```
int wl_list_empty(const struct wl_list *list)
```

wl_list_insert_list -

```
void wl_list_insert_list(struct wl_list *list, struct wl_list *other)
```

wl_array_init -

```
void wl_array_init(struct wl_array *array)
```

wl_array_release -

```
void wl_array_release(struct wl_array *array)
```

wl_array_add -

```
void* wl_array_add(struct wl_array *array, size_t size)
```

wl_array_copy -

```
int wl_array_copy(struct wl_array *array, struct wl_array *source)
```

wl_map_init -

```
void wl_map_init(struct wl_map *map, uint32_t side)
```

wl_map_release -

```
void wl_map_release(struct wl_map *map)
```

wl_map_insert_new -

```
uint32_t wl_map_insert_new(struct wl_map *map, uint32_t flags, void *data)
```

wl_map_insert_at -

```
int wl_map_insert_at(struct wl_map *map, uint32_t flags, uint32_t i, void *data)
```

wl_map_reserve_new -

```
int wl_map_reserve_new(struct wl_map *map, uint32_t i)
```

wl_map_remove -

```
void wl_map_remove(struct wl_map *map, uint32_t i)
```

wl_map_lookup -

```
void* wl_map_lookup(struct wl_map *map, uint32_t i)
```

wl_map_lookup_flags -

```
uint32_t wl_map_lookup_flags(struct wl_map *map, uint32_t i)
```

wl_map_for_each -

```
void wl_map_for_each(struct wl_map *map, wl_iterator_func_t func, void *data)
```

wl_log -

```
void wl_log(const char *fmt,...)
```

wl_list_init -

```
void wl_list_init(struct wl_list *list)
```

wl_list_insert -

```
void wl_list_insert(struct wl_list *list, struct wl_list *elm)
```

wl_list_remove -

```
void wl_list_remove(struct wl_list *elm)
```

wl_list_length -

```
int wl_list_length(const struct wl_list *list)
```

wl_list_empty -

```
int wl_list_empty(const struct wl_list *list)
```

wl_list_insert_list -

```
void wl_list_insert_list(struct wl_list *list, struct wl_list *other)
```

wl_array_init -

```
void wl_array_init(struct wl_array *array)
```

wl_array_release -

```
void wl_array_release(struct wl_array *array)
```

wl_array_add -

```
void* wl_array_add(struct wl_array *array, size_t size)
```

wl_array_copy -

```
int wl_array_copy(struct wl_array *array, struct wl_array *source)
```

5.2. Server API

Following is the Wayland library classes for the Server (*libwayland-server*). Note that most of the procedures are related with IPC, which is the main responsibility of the library.

wl_list - doubly-linked list

The list head is of "struct wl_list" type, and must be initialized using wl_list_init(). All entries in the list must be of the same type. The item type must have a "struct wl_list" member. This member will be initialized by wl_list_insert(). There is no need to call wl_list_init() on the individual item. To query if the list is empty in O(1), use wl_list_empty().

Let's call the list reference "struct wl_list foo_list", the item type as "item_t", and the item member as "struct wl_list link".

The following code will initialize a list: struct wl_list foo_list; struct item_t { int foo; struct wl_list link; }; struct item_t item1, item2, item3; wl_list_init(&foo_list); wl_list_insert(&foo_list, &item1.link); Pushes item1 at the head wl_list_insert(&foo_list, &item2.link); Pushes item2 at the head wl_list_insert(&item2.link, &item3.link); Pushes item3 after item2

The list now looks like [item2, item3, item1]

Will iterate the list in ascending order: item_t *item; wl_list_for_each(item, foo_list, link) { Do_something_with_item(item);

wl_listener - A single listener for Wayland signals.

wl_listener provides the means to listen for wl_signal notifications. Many Wayland objects use wl_listener for notification of significant events like object destruction.

Clients should create wl_listener objects manually and can register them as listeners to signals using wl_signal_add, assuming the signal is directly accessible. For opaque structs like wl_event_loop, adding a listener should be done through provided accessor methods. A listener can only listen to one signal at a time.

structwl_listeneryour_listener; your_listener.notify=your_callback_method; /*Directaccess*/ wl_signal_add(&some_object->destroy_signal,&your_listener); /*Accessoraccess*/ wl_event_loop*loop=...; wl_event_loop_add_destroy_listener(loop,&your_listener);

If the listener is part of a larger struct, wl_container_of can be used to retrieve a pointer to it:

*voidyour_listener(structwl_listener*listener,void*data) *{ *structyour_data*data=NULL; * *your_data=wl_container_of(listener,data,your_member_name); *} *

If you need to remove a listener from a signal, use #wl_list_remove.

*wl_list_remove(&your_listener.link); *

wl_signal

wl_signal - A source of a type of observable event.

Signals are recognized points where significant events can be observed. Compositors as well as the server can provide signals. Observers are wl_listener's that are added through wl_signal_add. Signals are emitted using wl_signal_emit, which will invoke all listeners until that listener is removed by #wl_list_remove (or whenever the signal is destroyed).

wl_listener for more information on using wl_signal

Methods for the respective classes.

wl_signal_init - Initialize a new wl_signal for use.

```
static void wl_signal_init(struct wl_signal *signal)
```

signal
: The signal that will be initialized

wl_signal_add - Add the specified listener to this signal.

```
static void wl_signal_add(struct wl_signal *signal, struct wl_listener *listener)
```

signal
: The signal that will emit events to the listener

listener
: The listener to add

wl_signal_get - Gets the listener struct for the specified callback.

```
static struct wl_listener * wl_signal_get(struct wl_signal *signal, wl_notify_func_t
 notify)
```

signal
: The signal that contains the specified listener

notify
: The listener that is the target of this search

Returns:
: the list item that corresponds to the specified listener, or NULL if none was found

wl_signal_emit - Emits this signal, notifying all registered listeners.

```
static void wl_signal_emit(struct wl_signal *signal, void *data)
```

signal
: The signal object that will emit the signal

data
: The data that will be emitted with the signal

wl_resource_post_event_array -

```
void wl_resource_post_event_array(struct wl_resource *resource, uint32_t opcode, union
 wl_argument *args)
```

wl_resource_post_event -

```
void wl_resource_post_event(struct wl_resource *resource, uint32_t opcode,...)
```

wl_resource_queue_event_array -

```
void wl_resource_queue_event_array(struct wl_resource *resource, uint32_t opcode, union
 wl_argument *args)
```

wl_resource_queue_event -

```
void wl_resource_queue_event(struct wl_resource *resource, uint32_t opcode,...)
```

wl_resource_post_error -

```
void wl_resource_post_error(struct wl_resource *resource, uint32_t code, const char
 *msg,...)
```

wl_client_flush -

```
void wl_client_flush(struct wl_client *client)
```

wl_client_get_display -

```
struct wl_display* wl_client_get_display(struct wl_client *client)
```

wl_client_create -

```
struct wl_client* wl_client_create(struct wl_display *display, int fd)
```

wl_client_get_credentials -

```
void wl_client_get_credentials(struct wl_client *client, pid_t *pid, uid_t *uid, gid_t
 *gid)
```

wl_client_get_object -

```
struct wl_resource* wl_client_get_object(struct wl_client *client, uint32_t id)
```

wl_client_post_no_memory -

```
void wl_client_post_no_memory(struct wl_client *client)
```

wl_resource_post_no_memory -

```
void wl_resource_post_no_memory(struct wl_resource *resource)
```

wl_resource_destroy -

```
void wl_resource_destroy(struct wl_resource *resource)
```

wl_resource_get_id -

```
uint32_t wl_resource_get_id(struct wl_resource *resource)
```

wl_resource_get_link -

```
struct wl_list* wl_resource_get_link(struct wl_resource *resource)
```

wl_resource_from_link -

```
struct wl_resource* wl_resource_from_link(struct wl_list *link)
```

wl_resource_find_for_client -

```
struct wl_resource* wl_resource_find_for_client(struct wl_list *list, struct wl_client
 *client)
```

wl_resource_get_client -

```
struct wl_client* wl_resource_get_client(struct wl_resource *resource)
```

wl_resource_set_user_data -

```
void wl_resource_set_user_data(struct wl_resource *resource, void *data)
```

wl_resource_get_user_data -

```
void* wl_resource_get_user_data(struct wl_resource *resource)
```

wl_resource_get_version -

```
int wl_resource_get_version(struct wl_resource *resource)
```

wl_resource_set_destructor -

```
void wl_resource_set_destructor(struct wl_resource *resource, wl_resource_destroy_func_t
 destroy)
```

wl_resource_instance_of -

```
int wl_resource_instance_of(struct wl_resource *resource, const struct wl_interface
 *interface, const void *implementation)
```

wl_resource_add_destroy_listener -

```
void wl_resource_add_destroy_listener(struct wl_resource *resource, struct wl_listener
 *listener)
```

wl_resource_get_destroy_listener -

```
struct wl_listener* wl_resource_get_destroy_listener(struct wl_resource *resource,
 wl_notify_func_t notify)
```

wl_client_add_destroy_listener -

```
void wl_client_add_destroy_listener(struct wl_client *client, struct wl_listener
 *listener)
```

wl_client_get_destroy_listener -

```
struct wl_listener* wl_client_get_destroy_listener(struct wl_client *client,
 wl_notify_func_t notify)
```

wl_client_destroy -

```
void wl_client_destroy(struct wl_client *client)
```

wl_display_create -

```
struct wl_display* wl_display_create(void)
```

wl_display_destroy -

```
void wl_display_destroy(struct wl_display *display)
```

wl_global_create -

```
struct wl_global* wl_global_create(struct wl_display *display, const struct wl_interface
 *interface, int version, void *data, wl_global_bind_func_t bind)
```

wl_global_destroy -

```
void wl_global_destroy(struct wl_global *global)
```

wl_display_get_serial -

```
uint32_t wl_display_get_serial(struct wl_display *display)
```

wl_display_next_serial -

```
uint32_t wl_display_next_serial(struct wl_display *display)
```

wl_display_get_event_loop -

```
struct wl_event_loop* wl_display_get_event_loop(struct wl_display *display)
```

wl_display_terminate -

```
void wl_display_terminate(struct wl_display *display)
```

wl_display_run -

```
void wl_display_run(struct wl_display *display)
```

wl_display_flush_clients -

```
void wl_display_flush_clients(struct wl_display *display)
```

wl_display_add_socket -

```
int wl_display_add_socket(struct wl_display *display, const char *name)
```

wl_display_add_destroy_listener -

```
void wl_display_add_destroy_listener(struct wl_display *display, struct wl_listener
 *listener)
```

wl_display_get_destroy_listener -

```
struct wl_listener* wl_display_get_destroy_listener(struct wl_display *display,
 wl_notify_func_t notify)
```

wl_resource_set_implementation -

```
void wl_resource_set_implementation(struct wl_resource *resource, const void
 *implementation, void *data, wl_resource_destroy_func_t destroy)
```

wl_resource_set_dispatcher -

```
void wl_resource_set_dispatcher(struct wl_resource *resource, wl_dispatcher_func_t
 dispatcher, const void *implementation, void *data, wl_resource_destroy_func_t destroy)
```

wl_resource_create -

```
struct wl_resource* wl_resource_create(struct wl_client *client, const struct
 wl_interface *interface, int version, uint32_t id)
```

wl_log_set_handler_server -

```
void wl_log_set_handler_server(wl_log_func_t handler)
```

wl_client_add_resource -

```
uint32_t wl_client_add_resource(struct wl_client *client, struct wl_resource *resource)
 WL_DEPRECATED
```

wl_client_add_object -

```
struct wl_resource * wl_client_add_object(struct wl_client *client, const struct
 wl_interface *interface, const void *implementation, uint32_t id, void *data)
 WL_DEPRECATED
```

wl_client_new_object -

```
struct wl_resource * wl_client_new_object(struct wl_client *client, const struct
 wl_interface *interface, const void *implementation, void *data) WL_DEPRECATED
```

wl_display_add_global -

```
struct wl_global * wl_display_add_global(struct wl_display *display, const struct
 wl_interface *interface, void *data, wl_global_bind_func_t bind) WL_DEPRECATED
```

wl_display_remove_global -

```
void wl_display_remove_global(struct wl_display *display, struct wl_global *global)
 WL_DEPRECATED
```

wl_display_add_shm_format -

```
void wl_display_add_shm_format(struct wl_display *display, uint32_t format)
```

wl_display_get_additional_shm_formats -

```
struct wl_array* wl_display_get_additional_shm_formats(struct wl_display *display)
```

wl_list_init -

```
void wl_list_init(struct wl_list *list)
```

wl_list_insert -

```
void wl_list_insert(struct wl_list *list, struct wl_list *elm)
```

wl_list_remove -

```
void wl_list_remove(struct wl_list *elm)
```

wl_list_length -

```
int wl_list_length(const struct wl_list *list)
```

wl_list_empty -

```
int wl_list_empty(const struct wl_list *list)
```

wl_list_insert_list -

```
void wl_list_insert_list(struct wl_list *list, struct wl_list *other)
```

wl_array_init -

```
void wl_array_init(struct wl_array *array)
```

wl_array_release -

```
void wl_array_release(struct wl_array *array)
```

wl_array_add -

```
void* wl_array_add(struct wl_array *array, size_t size)
```

wl_array_copy -

```
int wl_array_copy(struct wl_array *array, struct wl_array *source)
```

wl_map_init -

```
void wl_map_init(struct wl_map *map, uint32_t side)
```

wl_map_release -

```
void wl_map_release(struct wl_map *map)
```

wl_map_insert_new -

```
uint32_t wl_map_insert_new(struct wl_map *map, uint32_t flags, void *data)
```

wl_map_insert_at -

```
int wl_map_insert_at(struct wl_map *map, uint32_t flags, uint32_t i, void *data)
```

wl_map_reserve_new -

```
int wl_map_reserve_new(struct wl_map *map, uint32_t i)
```

wl_map_remove -

```
void wl_map_remove(struct wl_map *map, uint32_t i)
```

wl_map_lookup -

```
void* wl_map_lookup(struct wl_map *map, uint32_t i)
```

wl_map_lookup_flags -

```
uint32_t wl_map_lookup_flags(struct wl_map *map, uint32_t i)
```

wl_map_for_each -

```
void wl_map_for_each(struct wl_map *map, wl_iterator_func_t func, void *data)
```

wl_log -

```
void wl_log(const char *fmt,...)
```

wl_list_init -

```
void wl_list_init(struct wl_list *list)
```

wl_list_insert -

```
void wl_list_insert(struct wl_list *list, struct wl_list *elm)
```

wl_list_remove -

```
void wl_list_remove(struct wl_list *elm)
```

wl_list_length -

```
int wl_list_length(const struct wl_list *list)
```

wl_list_empty -

```
int wl_list_empty(const struct wl_list *list)
```

wl_list_insert_list -

```
void wl_list_insert_list(struct wl_list *list, struct wl_list *other)
```

wl_array_init -

```
void wl_array_init(struct wl_array *array)
```

wl_array_release -

```
void wl_array_release(struct wl_array *array)
```

wl_array_add -

```
void* wl_array_add(struct wl_array *array, size_t size)
```

wl_array_copy -

```
int wl_array_copy(struct wl_array *array, struct wl_array *source)
```