

Documentation 1.0

Wayland

The Wayland display server



Kristian Høgsberg

Documentation 1.0 Wayland

The Wayland display server

Edition 0

Author

Kristian Høgsberg

krh@bitplanet.net

Copyright © 2012 Kristian Høgsberg, Intel Corporation This material may only be distributed subject to the terms and conditions set forth in the GNU Free Documentation License (GFDL), V1.2 or later (the latest version is presently available at <http://www.gnu.org/licenses/fdl.txt>).

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.

1. Wayland Overview	1
1.1. Replacing X11	1
1.2. Make the compositing manager the display server	2
2. Wayland Architecture	5
2.1. X vs. Wayland Architecture	5
2.2. Wayland Rendering	7
2.3. Hardware Enabling for Wayland	8
3. The Wayland Protocol	9
3.1. Basic Principles	9
3.2. Code Generation	9
3.3. Wire Format	9
3.4. Interfaces	10
3.5. Connect Time	12
3.6. Security and Authentication	12
3.7. Creating Objects	12
3.8. Compositor	12
3.9. Surface	12
3.10. Input	12
3.11. Output	13
3.12. Data sharing between client (selection and drag and drop)	13
4. Wayland Library	15
4.1. Client API	15
5. Types of Compositors	17
5.1. System Compositor	17
5.2. Session Compositor	17
5.3. Embedding Compositor	17
A. Wayland Protocol Specification	19
A.1. wl_display - core global object	19
A.2. wl_registry - global registry object	21
A.3. wl_callback	22
A.4. wl_compositor - the compositor singleton	22
A.5. wl_shm_pool - a shared memory pool	22
A.6. wl_shm - shared memory support	23
A.7. wl_buffer - content for a wl_surface	25
A.8. wl_data_offer - offer to transfer data	25
A.9. wl_data_source - offer to transfer data	26
A.10. wl_data_device	27
A.11. wl_data_device_manager - data transfer interface	30
A.12. wl_shell	30
A.13. wl_shell_surface - desktop style meta data interface	30
A.14. wl_surface - an onscreen surface	36
A.15. wl_seat - seat	39
A.16. wl_pointer	40
A.17. wl_keyboard - keyboard input device	43
A.18. wl_touch - touch screen input device	45
A.19. wl_output - compositor output region	47
A.20. wl_region - region interface	50

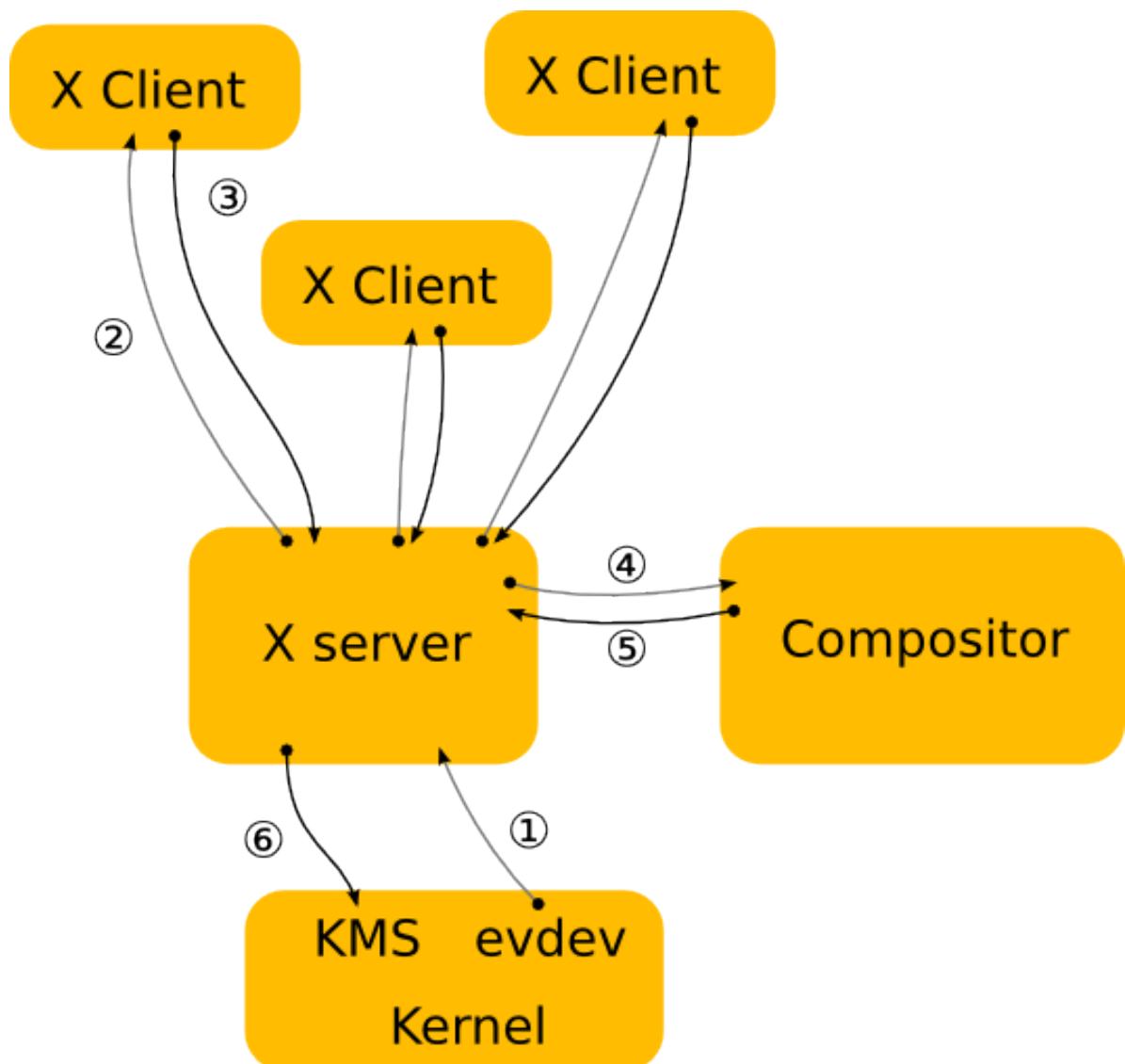
Wayland Overview

1. Wayland is a protocol for a new display server
2. Weston is the open source project implementing a wayland-based compositor

1.1. Replacing X11

In Linux and other Unix-like 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.

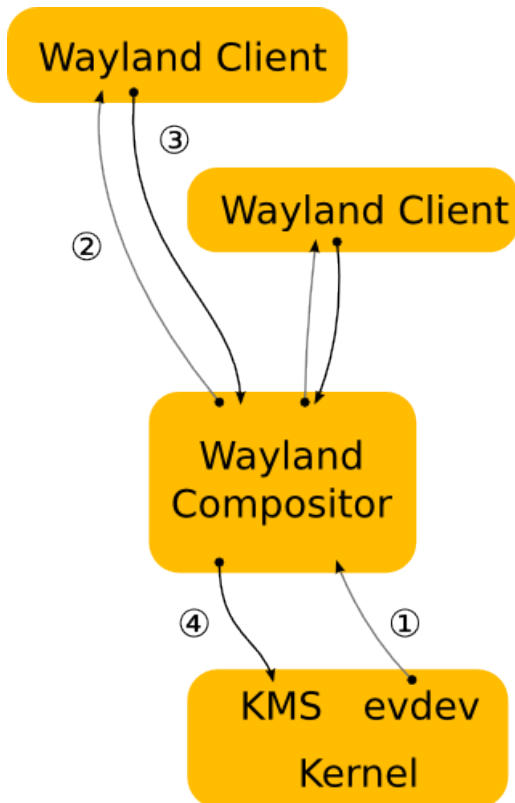
Over time, X developers came to understand the shortcomings of this approach and worked to split things up. Over the past several years, a lot of functionality has moved out of the X server and into client-side libraries or kernel drivers. One of the first components to move out was font rendering, with 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. Make the compositing manager 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.



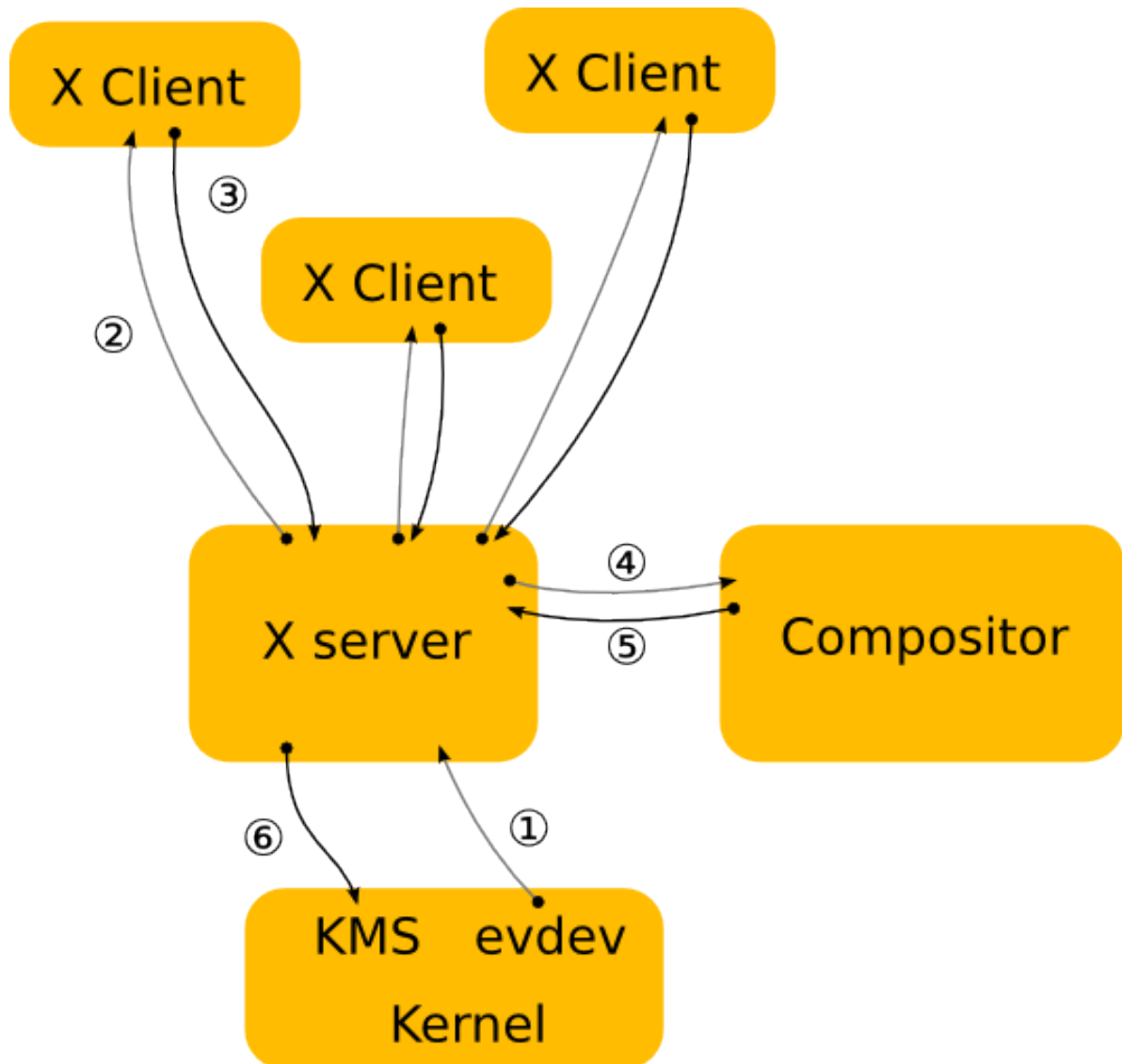
The figure above illustrates how Wayland clients interact with a Wayland server. Note that window management and composition are handled entirely in the server, significantly reducing complexity while marginally improving performance through reduced context switching. The resulting system is easier to 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).

Wayland Architecture

2.1. 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:

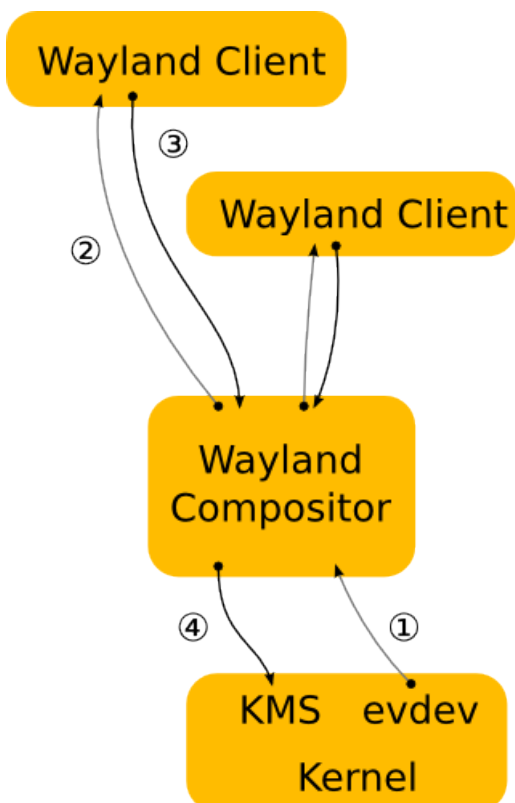


1. The kernel gets an event from an input device and sends it to X through the evdev input driver. The kernel does all the hard work here by driving the device and translating the different device specific event protocols to the linux evdev input event standard.
2. The X server determines which window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

2.2. 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 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.

2.3. 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 mode-setting 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 `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`.

The Wayland Protocol

3.1. Basic Principles

The wayland protocol is an asynchronous object oriented protocol. All requests are method invocations on some object. The request 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.

3.2. 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.

3.3. 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).

3.4. 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 provides 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 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

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. The objects will 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

Support for shared memory 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

wl_data_device_manager - data transfer interface

The wl_data_device_manager is a 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

wl_shell_surface - desktop style meta data interface

An interface implemented by a wl_surface. On 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. This is an image that is displayed on the screen. It has a location, size and pixel contents.

wl_seat - seat

A group of keyboards, pointer (mice, for example) 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

wl_keyboard - keyboard input device

wl_touch - touch screen input device

wl_output - compositor output region

An output describes part of the compositor geometry. The compositor work in the 'compositor co-ordinate 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 screen is hot plugged.

wl_region - region interface
Region.

3.5. Connect Time

- no fixed format connect block, the server emits a bunch of events at connect time
- presence events for global objects: output, compositor, input devices

3.6. 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.

3.7. Creating Objects

- client allocates object ID, uses range protocol
- server tracks how many IDs are left in current range, sends new range when client is about to run out.

3.8. Compositor

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

See [Section A.4](#), “*wl_compositor - the compositor singleton*” for the protocol description.

3.9. Surface

Created by the client.

See [Section A.14](#), “*wl_surface - an onscreen surface*” for the protocol description.

Needs a way to set input region, opaque region.

3.10. Input

Represents a group of input devices, including mice, keyboards. Has a keyboard and pointer focus. Global object. Pointer events are delivered in both screen coordinates and surface local coordinates.

See [Section A.15](#), “*wl_seat - seat*” 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 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 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.

3.11. Output

A output is a global object, advertised at connect time or as they come and go.

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

3.12. Data sharing between client (selection and drag and drop)

The Wayland 1.0 protocol provides its clients a mechanism for sharing data that allows the implementation of selection 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 through an fd that is passed through the protocol.

The next section explains the negotiation between data source and data offer objects. [Section 3.12.2, “Data devices”](#) explains how these objects are created and passed to different client using the `wl_data_device` interface, that implements selection 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”](#) 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).

3.12.1. 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

¹ <ftp://ftp.isi.edu/in-notes/iana/assignments/media-types/>

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 an `fd` 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 `fd` using the chosen mime type.

3.12.2. 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 selection or drag and drop.

3.12.2.1. 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.

3.12.2.2. Drag and Drop

A drag and drop operation is started using the `wl_data_device.start_drag` request. This request 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.

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.

Wayland Library

4.1. Client API

Following is the Wayland library classes for clients (*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_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.

And methods for the respective classes.

`wl_display_create_queue` - Create a new event queue for this display.

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

`wl_display_connect` - Connect to a Wayland display.

`wl_display_disconnect` - Close a connection to a Wayland display.

`wl_display_get_fd` - Get a display context's file descriptor.

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

`wl_display_dispatch_queue` - Dispatch events in an event queue.

`wl_display_dispatch_queue_pending` - Dispatch pending events in an event queue.

`wl_display_dispatch` - Process incoming events.

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

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

`wl_display_flush` - Send all buffered request on the display to the server.

`wl_event_queue_destroy` - Destroy an event queue.

`wl_proxy_create` - Create a proxy object with a given interface.

`wl_proxy_destroy` - Destroy a proxy object.

`wl_proxy_add_listener` - Set a proxy's listener.

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

`wl_proxy_set_user_data` - Set the user data associated with a proxy.

`wl_proxy_get_user_data` - Get the user data associated with a proxy.

`wl_proxy_get_id` - Get the id of a proxy object.

`wl_proxy_set_queue` - Assign a proxy to an event queue.

`wl_log_set_handler_client` -

Types of Compositors

5.1. System Compositor

- ties in with graphical boot
- hosts different types of session compositors
- lets us switch between multiple sessions (fast user switching, secure/personal desktop switching)
- multiseat
- linux implementation using libudev, egl, kms, evdev, cairo
- for fullscreen clients, the system compositor can reprogram the video scanout address to source from the client provided buffer.

5.2. Session Compositor

- nested under the system compositor. nesting is feasible because protocol is async, roundtrip would break nesting
- gnome-shell
- moblin
- compiz?
- kde compositor?
- text mode using vte
- rdp session
- fullscreen X session under wayland
- can run without system compositor, on the hw where it makes sense
- root window less X server, bridging X windows into a wayland session compositor

5.3. Embedding Compositor

X11 lets clients embed windows from other clients, or lets client 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 as 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.

- firefox embedding flash by being a special purpose compositor to the plugin

Appendix A. Wayland Protocol Specification

Copyright © 2008-2011 Kristian Høgsberg
Copyright © 2010-2011 Intel Corporation

Permission to use, copy, modify, distribute, and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of the copyright holders not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. The copyright holders make no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

THE COPYRIGHT HOLDERS DISCLAIM ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL THE COPYRIGHT HOLDERS BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

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 provided wl_callback object. Since requests are handled in-order, this can be used as a barrier to ensure all previous requests have been handled.

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 a 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 receives 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 provides 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 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 @name as the identifier.

wl_registry::bind arguments

name

Type: uint

unique number id for 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.

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 identifies 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 un-

til 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`

A.3.1. Events provided by `wl_callback`

A.3.1.1. `wl_callback::done`

`wl_callback::done` arguments

serial

Type: uint

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. The objects will share the same underlying mapped memory. Reusing the mapped memory avoids the setup/tear-down 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 wl_buffer from pool

Create a wl_buffer from the pool. The buffer is created a 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 pool.

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 fd passed when the pool was creating but using the new size.

wl_shm_pool::resize arguments

size

Type: int

A.6. wl_shm - shared memory support

Support for shared memory buffers.

A.6.1. Requests provided by wl_shm

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

This creates wl_shm_pool object, which can be used to create shared memory based wl_buffer objects. The server will mmap size bytes of the passed fd, to use as backing memory for then 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

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

invalid_format

Value: 0

invalid_stride

Value: 1

invalid_fd

Value: 2

A.6.3.2. wl_shm::format

wl_shm::format values

argb8888

Value: 0

xrgb8888

Value: 1

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. Use for feedback during drag and drop.

wl_data_offer::accept arguments

serial

Type: uint

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 fd (typically a pipe(7) file descriptor). The source client writes the data in the mime-type representation requested and then closes the fd. The receiving client reads from the read end of the pipe until EOF and then 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`

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

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

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.

wl_data_source::target arguments

mime_type

Type: string

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

Request for data from another client. Send the data as the specified mime-type over the passed fd, then close the fd.

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

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 nil) 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

A.10.1.2. `wl_data_device::set_selection`

`wl_data_device::set_selection` arguments

source

Type: object

serial

Type: uint

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

x

Type: fixed

y

Type: fixed

A.10.2.5. wl_data_device::drop

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 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`

`wl_data_device_manager::create_data_source` arguments

`id`

Type: `new_id`

A.11.1.2. `wl_data_device_manager::get_data_device`

`wl_data_device_manager::get_data_device` arguments

`id`

Type: `new_id`

`seat`

Type: `object`

A.12. wl_shell

A.12.1. Requests provided by `wl_shell`

A.12.1.1. `wl_shell::get_shell_surface`

`wl_shell::get_shell_surface` arguments

`id`

Type: `new_id`

`surface`

Type: `object`

A.13. wl_shell_surface - desktop style meta data interface

An interface implemented by a `wl_surface`. On 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

A.13.1.2. wl_shell_surface::move

wl_shell_surface::move arguments

seat

Type: object

serial

Type: uint

A.13.1.3. wl_shell_surface::resize

wl_shell_surface::resize arguments

seat

Type: object

serial

Type: uint

edges

Type: uint

A.13.1.4. wl_shell_surface::set_toplevel - make the surface a top level surface

Make the surface a toplevel window.

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. The flags argument controls overflow/clipping behaviour when the surface would intersect a screen edge, panel or such. And possibly whether the offset only determines the initial position or if the surface is locked to that relative position during moves.

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 `fullscreen_method` parameter.

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

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

Popup surfaces. Will switch an implicit grab into owner-events mode, and grab will continue after the implicit grab ends (button released). Once the implicit grab is over, 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.

TODO: Grab keyboard too, maybe just terminate on any click inside or outside the surface?

`wl_shell_surface::set_popup` arguments

seat
Type: object

serial

Type: uint

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

A request from the client to notify the compositor the maximized operation. 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 client will fill the fullscreen of the output it is bound to, except the panel area. This is the main difference between a maximized shell surface and a fullscreen shell surface.

[wl_shell_surface::set_maximized arguments](#)

output

Type: object

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

[wl_shell_surface::set_title arguments](#)

title

Type: string

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

The surface class identifies the general class of applications to which the surface belongs. The class is the file name of the applications .desktop file (absolute path if non-standard location).

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

`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 users 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`

`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

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 compositor how to deal with a conflict between the dimensions for the surface and the dimensions of the output. As a hint the compositor is free to ignore this parameter.

"default" The client has no preference on fullscreen behavior, policies are determined by compositor.

"scale" The client prefers scaling by the compositor. Scaling would always preserve surface's aspect ratio with surface centered on the output

"driver" The client wants to switch video mode to the smallest mode that can fit the client buffer. If the sizes do not match the compositor must add black borders.

"fill" The surface is centered on the output on the screen with no scaling. If the surface is of insufficient size the compositor must add black borders.

wl_shell_surface::fullscreen_method values

default
Value: 0

scale
Value: 1

driver
Value: 2

fill

Value: 3

A.14. wl_surface - an onscreen surface

A surface. This is an image that is displayed on the screen. It has a location, size and pixel contents.

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 the contents of a buffer into this surface. 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 other words, the x and y, and the width and height of the wl_buffer together 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 applies the pending wl_buffer as the new surface contents, and the size of the surface becomes the size of the wl_buffer. The wl_buffer is also kept as pending, until changed by wl_surface.attach or the wl_buffer is destroyed.

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, even if the wl_buffer is still pending for the next commit. In such case, the next commit does not change the surface contents. However, if the client destroys the wl_buffer before receiving wl_buffer.release, the surface contents become undefined immediately.

Only if wl_surface.attach is sent with a nil 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 (or if pending buffer is none, the current buffer as updated in-place) on the next wl_surface.commit will be different from the current buffer, and needs to be repainted. The pending buffer can be set by wl_surface.attach. The compositor ignores the parts of the damage that fall outside of the surface.

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

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.

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 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 nil `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.

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 nil `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 the pending `wl_buffer` is none, 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.14.2. 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.

`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** - seat

A group of keyboards, pointer (mice, for example) 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.

[wl_seat::get_pointer arguments](#)

id

Type: new_id

A.15.1.2. **wl_seat::get_keyboard** - return pointer object

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

[wl_seat::get_keyboard arguments](#)

id

Type: new_id

A.15.1.3. **wl_seat::get_touch** - return pointer object

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

[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 wl_seat_caps_mask enum containing the complete set of capabilities this seat has.

[wl_seat::capabilities arguments](#)

capabilities

Type: uint

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

`wl_pointer`

keyboard

Value: 2

`wl_keyboard`

touch

Value: 4

`wl_touch`

A.16. `wl_pointer`

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 $(x, y) - (\text{hotspot_x}, \text{hotspot_y})$, where (x, y) are the coordinates of the pointer location.

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

surface
Type: object

hotspot_x
Type: int

hotspot_y
Type: int

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.

wl_pointer::enter arguments

serial
Type: uint

surface
Type: object

surface_x
Type: fixed

surface_y
Type: fixed

A.16.2.2. wl_pointer::leave - leave event

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_[xy] are the location relative to the focused surface.

wl_pointer::motion arguments

time
Type: uint

surface_x

Type: fixed

surface_y

Type: fixed

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 pointer_focus event.

wl_pointer::button arguments

serial

Type: uint

time

Type: uint

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

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

button is not pressed

pressed

Value: 1

button is pressed

A.16.3.2. wl_pointer::axis - axis types

wl_pointer::axis values

vertical_scroll

Value: 0

horizontal_scroll

Value: 1

A.17. wl_keyboard - keyboard input device

A.17.1. Events provided by wl_keyboard

A.17.1.1. 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.17.1.2. wl_keyboard::enter

[wl_keyboard::enter arguments](#)

serial

Type: uint

surface

Type: object

keys

Type: array

A.17.1.3. wl_keyboard::leave

[wl_keyboard::leave arguments](#)

serial

Type: uint

surface

Type: object

A.17.1.4. wl_keyboard::key - key event

A key was pressed or released.

[wl_keyboard::key arguments](#)

serial

Type: uint

time

Type: uint

key

Type: uint

state

Type: uint

A.17.1.5. 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.17.2. Enums provided by wl_keyboard

A.17.2.1. wl_keyboard::keymap_format - keyboard mapping format

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

wl_keyboard::keymap_format values

xkb_v1

Value: 1

A.17.2.2. 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 - touch screen input device

A.18.1. Events provided by wl_touch

A.18.1.1. wl_touch::down

wl_touch::down arguments

serial

Type: uint

time

Type: uint

surface

Type: object

id

Type: int

x

Type: fixed

y

Type: fixed

A.18.1.2. wl_touch::up

wl_touch::up arguments

serial

Type: uint

time

Type: uint

id

Type: int

A.18.1.3. wl_touch::motion

wl_touch::motion arguments

time

Type: uint

id

Type: int

x

Type: fixed

y

Type: fixed

A.18.1.4. wl_touch::frame - end of touch frame event

Indicates the end of a contact point list.

A.18.1.5. `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.

A.19. `wl_output` - compositor output region

An output describes part of the compositor geometry. The compositor work 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 screen is hot plugged.

A.19.1. Events provided by `wl_output`

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

`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.

`wl_output::mode` arguments

flags

Type: uint

mask of `wl_output_mode` flags

width

Type: int

width of the mode in pixels

height

Type: int

height of the mode in pixels

refresh

Type: int

vertical refresh rate in mHz

A.19.2. Enums provided by `wl_output`

A.19.2.1. `wl_output::subpixel`

`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 - values for the flags bitfield in 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

Region.

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