The Wayland Protocol

Drew DeVault

Contents

1. *Introduction*

Wayland is the next-generation display server for Unix-like systems, designed and built by the alumni of the venerable Xorg server, and is the best way to get your application windows onto your user's screens. Readers who have worked with X11 in the past will be pleasantly surprised by Wayland's improvements, and those who are new to graphics on Unix will find it a flexible and powerful system for building graphical applications and desktops.

This book will help you establish a firm understanding of the concepts, design, and implementation of Wayland, and equip you with the tools to build your own Wayland client and server applications. Over the course of your reading, we'll build a mental model of Wayland and establish the rationale that went into its design. Within these pages you should find many "aha!" moments as the intuitive design choices of Wayland become clear, which should help to keep the pages turning. Welcome to the future of open source graphics!

1.1 About the author

In the words of Preston Carpenter, a close collaborator of Drew's:

> Drew DeVault got his start in the Wayland world by building sway, a clone of the popular tiling window manager i3. It is now the most popular tiling Wayland compositor by any measure: users, commit count, and influence. Following its success, Drew gave back to the Wayland community by starting wlroots: unopinionated, composable modules for building a Wayland compositor. Today it is the foundation for dozens of independent compositors, and Drew is one of the foremost experts in Wayland.

1.2 Goals & target audience

Our goal is for you to come away from this book with an understanding of the Wayland protocol and its high-level usage. You should have a solid understanding of everything in the core Wayland protocol, as well as the knowledge necessary to evaluate and implement the various protocol extensions necessary for its productive use. Primarily, this book uses the concerns of a Wayland client to frame its presentation of Wayland. However, it should provide some utility for those working on Wayland compositors as well.

The free desktop ecosystem is complex and built from many discrete parts. We are going to discuss these pieces very little — you won't find information here about leveraging libdrm in your Wayland compositor, or using libinput to process evdev events. Thus, this book is not a comprehensive guide for building Wayland compositors. We're also not going to talk about drawing technologies which are useful for Wayland clients, such as Cairo, Pango, GTK+, and so on, and thus neither is this a robust guide for the practical Wayland client implementation. Instead, we focus only on the particulars of Wayland.

This book only covers the protocol and libwayland. If you are writing a client and are already familiar with your favorite user interface rendering library, bring your own pixels and we'll help you display them on Wayland. If you already have an understanding of the technologies required to operate displays and input devices for your compositor, this book will help you learn how to talk to clients.

1.3 High Level Design

Your computer has *input* and *output* devices, which respectively are responsible for receiving information from you and displaying information to you. These input devices take the form of, for example:

- Keyboards
- Mice
- Touchpads
- Touch screens
- Drawing tablets

Your output devices generally take the form of displays, on your desk or your laptop or mobile device. These resources are shared between all of your applications, and the role of the **Wayland compositor** is to dispatch input events to the appropriate **Wayland client** and to display their windows in their appropriate place on your outputs. The process of bringing together all of your application windows for display on an output is called *compositing* - and thus we call the software which does this the *compositor*.

1.3.1 In Practice

There are many distinct software components in the desktop ecosystem. There are tools like mesa for rendering (and each of its drivers), the Linux KMS/DRM subsystem, buffer allocation with GBM, the userspace libdrm library, libinput and evdev, and much more still. Don't worry — expertise with most of these systems is not required for understanding Wayland, and in any case are largely beyond the scope of this book. In fact, the Wayland protocol is quite conservative and abstract, and a Wayland-based desktop could easily be built & run most applications without implicating any of this software. That being said, a surface-level understanding of what these pieces are and how they work is useful. Let's start from the bottom and work our way up.

1.3.2 The hardware

A typical computer is equipped with a few important pieces of hardware. Outside of the box, we have your displays, keyboard, mouse, perhaps some speakers and a cute USB cup warmer. There are several components *inside* the box for interfacing with these devices. Your keyboard and mouse, for example, are probably plugged into USB ports, for which your system has a dedicated USB controller. Your displays are plugged into your GPU.

These systems have their own jobs and state. For example, your GPU has state in the form of memory for storing pixel buffers in, and jobs like *scanning out* these buffers to your displays. Your GPU also provides a processor which is specially tuned to be good at highly parallel jobs (such as calculating the right color for each of the 2,073,600 pixels on a 1080p display), and bad at everything else. The USB controller has the unenviable job of implementing the legendarily dry USB specification for receiving input events from your keyboard, or instructing your coaster to assume a temperature carefully selected to at once avoid lawsuits and frustrate you with cold coffee.

At this level, your hardware has little concept of what applications are running on your system. The hardware provides an interface with which it can be commanded to perform work, and does what it's told — regardless of who tells it so. For this reason, only one component is allowed to talk to it…

1.3.3 The kernel

This responsibility falls onto the kernel. The kernel is a complex beast, so we'll focus on only the parts which are relevant to Wayland. Linux's job is to provide an abstraction over your hardware, so that they can be safely accessed by *userspace* — where our Wayland compositors run. For graphics, this is called **DRM**, or *Direct Rendering Manager*, for efficiently tasking the GPU with work from userspace. An important subsystem of DRM is **KMS**, or *Kernel Mode Setting*, for enumerating your displays and setting properties such as their selected resolution (also known as their "mode"). Input devices are abstracted through an interface called **evdev**.

Most kernel interfaces are made available to userspace by way of special files in `/dev`. In the case of DRM, these files are in `/dev/dri/`, usually in the form of a primary node (e.g. `card0`) for

privileged operations like modesetting, and a render node (e.g. `renderD128`), for unprivileged operations like rendering or video decoding. For evdev, the "device nodes" are `/dev/input/event*`.

1.3.4 Userspace

Now, we enter userspace. Here, applications are isolated from the hardware and must work via the device nodes provided by the kernel.

libdrm

Most Linux interfaces have a userspace counterpart which provides a pleasant(ish) C API for working with these device nodes. One such library is libdrm, which is the userspace portion of the DRM subsystem. libdrm is used by Wayland compositors to do modesetting and other DRM operations, but is generally not used by Wayland clients directly.

Mesa

Mesa is one of the most important parts of the Linux graphics stack. It provides, among other things, vendor-optimized implementations of OpenGL (and Vulkan) for Linux and the **GBM** (*Generic Buffer Management*) library — an abstraction on top of libdrm for allocating buffers on the GPU. Most Wayland compositors will use both GBM and OpenGL via Mesa, and most Wayland clients will use at least its OpenGL or Vulkan implementations.

libinput

Like libdrm abstracts the DRM subsystem, libinput provides the userspace end of evdev. It's responsible for receiving input events from the kernel from your various input devices, decoding them into a usable form, and passing them on to the Wayland compositor. The Wayland compositor requires special permissions to use the evdev files, forcing Wayland clients to go through the compositor to receive input events — which, for example, prevents keylogging.

(e)udev

Dealing with the appearance of new devices from the kernel, configuring permissions for the resulting device nodes in `/dev`, and sending word of these changes to applications running on your system, is a responsibility that falls onto userspace. Most systems use udev (or eudev, a fork) for this purpose. Your Wayland compositor uses udev to enumerate input devices and GPUs, and to receive notifications when new ones appear or old ones are unplugged.

xkbcommon

XKB, short for X keyboard, is the original keyboard handling subsystem of the Xorg server. Several years ago, it was extracted from the Xorg tree and made into an independent library for keyboard

handling, and it no longer has any practical relationship with X. Libinput (along with the Wayland compositor) delivers keyboard events in the form of scancodes, whose precise meaning varies from keyboard to keyboard. It's the responsibility of xkbcommon to translate these scan codes into meaningful and generic key "symbols" — for example, converting 65 to `XKB_KEY_Space`. It also contains a state machine which knows that pressing "1" while shift is held emits "!".

pixman

A simple library used by clients and compositors alike for efficiently manipulating pixel buffers, doing math with intersecting rectangles, and performing other similar pixel manipulation tasks.

libwayland

libwayland the most commonly used implementation of the Wayland protocol, is written in C, and handles much of the low-level wire protocol. It also provides a tool which generates high-level code from Wayland protocol definitions (which are XML files). We will be discussing libwayland in detail in chapter 1.3, and throughout this book.

...and all the rest.

Each of the pieces mentioned so far are consistently found throughout the Linux desktop ecosystem. Beyond this, more components exist. Many graphical applications don't know about Wayland at all, choosing instead to allow libraries like GTK+, Qt, SDL, and GLFW — among many others — to deal with it. Many compositors choose software like wlroots to abstract more of their responsibilities, while others implement everything in-house.

1.4 What's in the Wayland package

When you install "wayland" in your Linux distribution, you'll most likely end up with the freedesktop.org distribution of libwayland-client, libwayland-server, wayland-scanner, and wayland.xml. Respectively, these will probably be in `/usr/lib` & `/usr/include`, `/usr/bin`, and `/usr/share/wayland/`. This package represents the most popular *implementation* of the Wayland protocol, but it is not the only one. Chapter 3 covers this implementation of Wayland in detail; the rest of the book is equally applicable to any implementation.

1.4.1 wayland.xml

Wayland protocols are defined by XML files. If you locate and open "wayland.xml" in your favorite text editor, you will find the XML specification for the "core" Wayland protocol. This is a high-level protocol — built on top of the wire protocol that we'll discuss in the next chapter. Most of this book is devoted to explaining this file.

1.4.2 wayland-scanner

The ”wayland-scanner” tool is used to process these XML files and generate code from them. The most common implementation is the one you're examining now, and it can be used to generate C headers & glue code from XML files like wayland.xml. Other scanners exist for other programming languages, notably wayland-rs (Rust), waymonad-scanner (Haskell), and more.

1.4.3 libwayland

The two libraries, libwayland-client and libwayland-server, include an implementation of the wire protocol for each end of the connection. Some common utilities are offered for working with Wayland data structures, a simple event loop, and so on. Additionally, these libraries contain a precompiled copy of the core Wayland protocol as generated by `wayland-scanner`.

2. *Protocol Design*

The Wayland protocol is built from several layers of abstraction. It starts with a basic wire protocol format, which is a stream of messages decodable with interfaces agreed upon in advance. Then we have higher level procedures for enumerating interfaces, creating resources which conform to these interfaces, and exchanging messages about them — the Wayland protocol and its extensions. On top of this we have some broader patterns which are frequently used in Wayland protocol design. We'll cover all of these in this chapter.

Let's work our way from the bottom-up.

Note

> If you're just going to use libwayland, this chapter is optional — feel free to skip to chapter 2.2.

2.1 Wire Protocol Basics

The wire protocol is a stream of 32-bit values, encoded with the host's byte order (e.g. little-endian on x86 family CPUs). These values represent the following primitive types:

Type	Description
int, uint	32-bit signed or unsigned integer
fixed	24.8-bit signed fixed-point number
object	32-bit object ID
new_id	32-bit object ID which allocates that object when received.

Table 2.1: The Wayland protocol's primitive types

In addition to these primitives, the following other types are used:

string A string, prefixed with a 32-bit integer specifying its length (in bytes), followed by the string contents and a `NUL` terminator, padded to 32 bits with undefined data. The encoding is not specified, but in practice UTF-8 is used.

array A blob of arbitrary data, prefixed with a 32-bit integer specifying its length (in bytes), then the verbatim contents of the array, padded to 32 bits with undefined data.

fd 0-bit value on the primary transport, but transfers a file descriptor to the other end using the ancillary data in the Unix domain socket message (`msg_control`).

enum A single value (or bitmap) from an enumeration of known constants, encoded into a 32-bit integer.

2.1.1 Messages

The wire protocol is a stream of messages built with these primitives. Every message is an event (in the case of server to client messages) or request (client to server) which acts upon an *object*.

The message header is two words. The first word is the affected object ID. The second is two 16-bit values; the upper 16 bits are the size of the message (including the header itself) and the lower 16 bits are the event or request opcode. The message arguments follow, based on a message signature agreed upon in advance by both parties. The recipient looks up the object ID's interface and the event or request defined by its opcode to determine the signature and nature of the message.

To understand a message, the client and server have to establish the objects in the first place. Object ID 1 is pre-allocated as the Wayland display singleton, and can be used to bootstrap other objects. We'll discuss this in chapter 4. The next chapter goes over what an interface is, and how requests and events work, assuming you've already negotiated an object ID. Speaking of which…

2.1.2 Object IDs

When a message comes in with a `new_id` argument, the sender allocates an object ID for it (the interface used for this object is established through additional arguments, or agreed upon in advance for that request/event). This object ID can be used in future messages, either as the first word of the header, or as an `object_id` argument. The client allocates IDs in the range of `[1, 0xFEFFFFFF]`, and the server allocates IDs in the range of `[0xFF000000, 0xFFFFFFFF]`. IDs begin at the lower end of this bound and increment with each new object allocation.

An object ID of `0` represents a null object; that is, a non-existent object or the explicit lack of an object.

2.1.3 Transports

To date all known Wayland implementations work over a Unix domain socket. This is used for one reason in particular: file descriptor messages. Unix sockets are the most practical transport capable of transferring file descriptors between processes, and this is necessary for large data transfers (keymaps, pixel buffers, and clipboard contents being the main use-cases). In theory, a different transport (e.g. TCP) is possible, but someone would have to figure out an alternative way of transferring bulk data.

To find the Unix socket to connect to, most implementations just do what libwayland does:

1. If `$WAYLAND_SOCKET` is set, interpret it as a file descriptor number on which the connection is already established, assuming that the parent process configured the connection for us.
2. If `$WAYLAND_DISPLAY` is set, concat with `$XDG_RUNTIME_DIR` to form the path to the Unix socket.
3. Assume the socket name is `wayland-0` and concat with `$XDG_RUNTIME_DIR` to form the path to the Unix socket.
4. Give up.

2.2 Interfaces, requests, and events

The Wayland protocol works by issuing *requests* and *events* that act on *objects*. Each object has an *interface* which defines what requests and events are possible, and the *signature* of each. Let's consider an example interface: `wl_surface`.

2.2.1 Requests

A surface is a box of pixels that can be displayed on-screen. It's one of the primitives we build things like application windows out of. One of its *requests* is "damage", which the client uses to

indicate that some part of the surface has changed and needs to be redrawn. Here's an annotated example of a "damage" message on the wire (in hexadecimal):

Message bytes	Description		
0000000A	Object ID	id	10
0018xxxx	Message length	int (MSW)	24
xxxx0002	Request opcode	int (LSW)	2
00000000	X coordinate	int	0
00000000	Y coordinate	int	0
00000100	Width	int	256
00000100	Height	int	256

Table 2.2: MSW: Most significant word; LSW: Least significant word; of the same 32-bit value

This is a snippet of a session — the surface was allocated earlier and assigned an ID of 10. When the server receives this message, it looks up the object with ID 10 and finds that it's a `wl_surface` instance. Knowing this, it looks up the signature for the request with opcode 2. It then knows to expect four integers as the arguments, and it can decode the message and dispatch it for processing internally.

2.2.2 Events

Requests are sent from the client to the server. The server can also send messages back: events. One event that the server can send regarding a `wl_surface` is "enter", which it sends when that surface is being displayed on a specific output (the client might respond to this, for example, by adjusting its scale factor for a HiDPI display). Here's an example of such a message:

Message bytes	Description		
0000000A	Object ID	id	10
000Bxxxx	Message length	int (MSW)	12
xxxx0000	Request opcode	int (LSW)	0
00000005	Output	id	5

This message references another object, by its ID: the `wl_output` object which the surface is being shown on. The client receives this and dances to a similar tune as the server did. It looks up object 10, associates it with the `wl_surface` interface, and looks up the signature of the event corresponding to opcode 0. It decodes the rest of the message accordingly (looking up the `wl_output` with ID 5 as well), then dispatches it for processing internally.

2.2.3 Interfaces

The interfaces which define the list of requests and events, the opcodes associated with each, and the signatures with which you can decode the messages — are agreed upon in advance. I'm sure you're dying to know how — let's end the suspense.

2.3 The high-level protocol

In chapter 1.3, I mentioned that `wayland.xml` is probably installed with the Wayland package on your system. Find and pull up that file now in your favorite text editor. It's through this file, and others like it, that we define the interfaces supported by a Wayland client or server.

Each interface is defined in this file, along with its requests and events, and their respective signatures. We use XML, everyone's favorite file format, for this purpose. Let's look at the examples we discussed in the previous chapter for `wl_surface`. Here's a sample:

```
<interface name="wl_surface" version="4">
  <request name="damage">
    <arg name="x" type="int" />
    <arg name="y" type="int" />
    <arg name="width" type="int" />
    <arg name="height" type="int" />
  </request>

  <event name="enter">
    <arg name="output" type="object" interface="wl_output" />
  </event>
</interface>
```

Note

> I've trimmed this snippet for brevity, but if you have the `wayland.xml` file in front of you, seek out this interface and examine it yourself — included is additional documentation explaining the purpose and precise semantics of each request and event.

When processing this XML file, we assign each request and event an opcode in the order that they appear (both numbered from zero and incrementing independently). Combined with the list of arguments, you can decode the request or event when it comes in over the wire, and based on the documentation shipped in the XML file you can decide how to program your software to behave accordingly. This usually comes in the form of code generation — we'll talk about how libwayland does this in chapter 3.

Starting from chapter 4, most of the remainder of this book is devoted to explaining this file, as well as some supplementary protocol extensions.