JPEG XL Decoding Support using jxl-rs

This Document is Public

Authors: helmut@janschka.com

November 2025

One-page overview

Add JPEG XL (JXL) image decoding support to Chromium using jxl-rs, a pure Rust decoder. This enables native rendering of .jxl images which offer 30-50% better compression than JPEG, lossless JPEG transcoding, progressive decoding, and animation support. Using a Rust decoder provides memory safety guarantees for image parsing, reducing attack surface compared to C++ alternatives.

Platforms

Mac, Windows, Linux, Chrome OS, Android, Android WebView.

Team

- Author: Helmut Januschka

Bug

https://issues.chromium.org/issues/462919304

CL

https://chromium-review.googlesource.com/c/chromium/src/+/7184969

.

Code affected

- Blink image decoders - New JXLImageDecoder class
- Media build configuration - ENABLE_JXL_DECODER flag
- CC paint/tiles - ImageType enum, decode cache
- Third-party Rust crates - jxl-rs and dependencies
- Third-party jxl-rs C API - FFI wrapper for Rust decoder

Design

Architecture Overview

This implementation adds JPEG XL decoding support to Chromium using jxl-rs, a pure Rust decoder.
JXL is integrated at the Blink ImageDecoder level (following the same pattern as AVIF, WebP, GIF, and animated PNG) rather than through Skia's codec API. This integration point is necessary because**:**

1. **Animation support**: JXL animations require frame-by-frame decoding with precise timing control and frame caching. The Blink ImageDecoder API provides `DecodeFrameCount()`, `FrameDurationAtIndex()`, `FrameTimestampAtIndex()`, and `ClearCacheExceptFrame()` specifically designed for animated formats. While PNG's animation support (APNG) is relatively simple, JXL animations can have complex frame dependencies and variable frame durations that benefit from Blink's animation framework.

2. **Progressive rendering**: JXL supports progressive decoding where the image quality improves as more data arrives. Blink's ImageDecoder API supports this through `OnSetData()` and `FrameIsReceivedAtIndex()`, allowing images to render incrementally as data streams over the network. Skia's codec API is primarily designed for complete images.

3. **Decoder state management**: Unlike PNG (which decodes row-by-row), JXL uses a stateful decoding process that requires coordination between metadata parsing and frame decoding. The Blink ImageDecoder pattern provides explicit control over decoder lifecycle through `DecodeSize()`, `DecodeFrameCount()`, and `Decode()` phases that match JXL's architecture.

Why PNG could use Skia but JXL cannot: PNG's integration into Skia was feasible because: (a) PNG animations (APNG) have simple frame timing and no inter-frame dependencies, (b) PNG's row-by-row decoding maps cleanly to Skia's incremental codec APIs, and (c) PNG is used across many Skia clients (Android, Flutter, etc.) making Skia-level integration beneficial. JXL's more complex animation model and Chromium-specific requirements make Blink-level integration more appropriate.

The architecture consists of two main components

Feature Support

JPEG XL offers advanced imaging capabilities beyond basic compression. Support status for key features:

Feature	Status	Implementation Details
Animation	✅ Supported	Frame-by-frame decoding via `DecodeFrameCount()`,
Progressive rendering	🛑 Unsupported for now	Once jxl-rs supports it. Progressive decoding will use jxl-rs's internal sophisticated non-separable upscaling for DC data, matching the behavior of progressive legacy JPEGs (libjpeg-turbo). This avoids intermediate buffers in Chromium. Adding support tracked by progress bug.
Wide gamut color	✅ Supported	ICC profile extraction + F32→F16 pipeline
10/12-bit color depth	✅ Supported	All bit depths (8/10/12/16-bit)
HDR (PQ/HLG/21496-1)	✅ Supported	All HDR content (PQ/HLG) supported via ICC profile extraction + F32→F16 pipeline. Gainmap support is now available in an upstream PR. We will integrate this once merged. PR
Alpha channel	✅ Supported	RGB and alpha decoded separately, interleaved
Lossless JPEG recompression	✅ Supported	jxl-rs decodes both lossless

Component 1: JXLImageDecoder (C++)

Location: `third_party/blink/renderer/platform/image-decoders/jxl/`

A new `JXLImageDecoder` class that extends Blink's `ImageDecoder` base class:

- Handles integration with Chromium's image loading pipeline
- Manages progressive decoding and animation state
- Converts decoded pixels to Skia-compatible formats
- Implements frame caching for smooth animation playback

Component 2: jxl-rs C++ Bridge (Rust + cxx codegen)

Location: `third_party/jxl-rs/src/`

A type-safe C++/Rust bridge using the cxx library.

What cxx generates:

- **C++ header**: `lib.rs.h` with type definitions (`JxlRsDecoder`, `JxlRsBasicInfo`, `JxlRsStatus`, etc.)
- **FFI implementation**: `lib.rs.cc` with glue code between C++ and Rust
- **Type-safe wrappers**: `rust::Box<T>` for ownership, `rust::Slice<T>` for zero-copy array views

Key responsibilities:

- **Decoder lifecycle management**:
  - Factory: `jxl_rs_decoder_create()` returns `rust::Box<JxlRsDecoder>`
  - State machine: `set_input()` → `process()` → `get_basic_info()` / `get_pixels()`
  - Automatic cleanup: `rust::Box` handles destruction

- **Pixel format conversion**:
  - Internal: Always `f32` RGB (separate from alpha) in linear color space
  - Output formats:
    - `Rgba8`: f32 [0.0–1.0] → u8 [0–255] with clamping

    - `Rgba16`: f32 [0.0–1.0] → u16 [0–65535] with clamping
    - `RgbaF32`: Direct f32 (preserves HDR values >1.0)

- **Alpha channel handling**:
    - Decodes RGB and alpha into separate buffers (per JXL spec)
    - Interleaves during `get_pixels()`: `[R,G,B,A,R,G,B,A,...]`
    - Defaults to opaque (255/1.0) if no alpha channel

- **Panic safety**:
    - Chromium compiles all Rust with `-Cpanic=abort`, so panics terminate the process immediately
    - No `catch_unwind()` needed - panics are prevented upstream in jxl-rs

Data Flow

- Blink receives image data → JXLImageDecoder::OnSetData()
- Signature check via jxl_rs_signature_check() (12-byte header)
- Decoder processes input → emits BasicInfo, Frame, FullImage events
- ICC profile extraction → SetEmbeddedColorProfile() for color management
- Pixel format selection based on image characteristics:
    - **High bit depth (10/12/16-bit), HDR (PQ/HLG), or wide gamut (Display P3, Rec.2020)**:
        - jxl-rs outputs F32 linear RGB → convert to F16
        - Preserves extended range values >1.0 for HDR
        - No clamping during conversion
    - **Standard 8-bit sRGB**:
        - jxl-rs outputs F32 linear RGB → convert to U8
        - Clamp [0.0-1.0] → map to [0-255]
-
- Pixels written to Skia ImageFrame (kRGBA_F16 or kN32 format)
- For animations: frame durations extracted, caching prevents re-decode

Key Design Decisions

Pure Rust decoder: Uses jxl-rs instead of libjxl (C++) for memory safety and reduced attack surface in image parsing code.

HDR support: HDR images (PQ/HLG transfer functions) are decoded with ICC profile extraction and extended range values preserved in f16 format. Tone mapping is handled dynamically by Chromium's color management system (not at decode time), allowing content to adapt to display capabilities and CSS attributes (e.g., https://mdn.github.io/dom-examples/dynamic-range-limit/).

Why jxl-rs

- **Upstream alignment:** jxl-rs is maintained in the same GitHub organization/namespace as **libjxl**, the official reference implementation. This suggests tighter coordination with the core JPEG XL maintainers and improves long-term compatibility with the evolving spec.
- **Browser-proven:** It is the same Rust JPEG XL implementation already integrated in **Firefox**, demonstrating production readiness for browser environments.
- **Active development:** jxl-rs has higher recent activity, more frequent updates, and faster turnaround on fixes compared to jxl-oxide.
- **Better feature and performance maturity:** It more closely tracks the reference implementation, including optimizations and optional features.
- **Integration benefits:** Its API surface and design fit well with Chromium's Rust FFI patterns and error-handling expectations.

Metrics

Success metrics

Functional parity:

- JXL images render correctly across all supported bit depths (8/10/12/16-bit)

- Animations playback smoothly with correct timing and frame recovery under load
- Progressive rendering works as data streams in
- Color management correctly handles wide gamut (Display P3, Rec.2020) via ICC profiles
- HDR content (PQ/HLG) displays correctly using f32 pipeline
- No crashes from malformed files (validated via fuzzing)

Usage tracking:

WebDX use counter: `WebFeature::kJXLImage` / `WebDXFeature::kJxl`

- Tracks when JXL images are successfully decoded and displayed
- Incremented when image size becomes available (metadata parsed)
- Visible in chrome://histograms and chromestatus.com metrics
- Helps measure adoption and justify feature value

Performance targets:

- Match libjxl's performance
- Automate periodic tests of performance, to track progress over time
- Memory usage within acceptable limits (monitored via `max_decoded_bytes`)
- No regressions in page load metrics (LCP, FCP)

Regression metrics

Performance:

- `Blink.ImageDecoders.DecodeTime` - JXL decode timing vs. other formats
- `Renderer4.ImageDecodeTaskDurationUs.Jxl.Gpu` - JXL-specific decode metric
- Web Vitals (LCP, FCP) - Page load impact

Memory:

- `Memory.Renderer.PrivateMemoryFootprint` - Renderer memory footprint
- Decoder enforces `max_decoded_bytes` limit to prevent OOM

Binary size:

- ~3-5 MB added (release APK, compressed)

Experiments

Build Flag

- `enable_jxl_decoder = use_blink` in `media/media_options.gni`

- Can be disabled per-platform

Runtime Feature (Kill-Switch)

Note: We cannot use `runtime_enabled_features.json5` because RuntimeEnabledFeatures don't support conditional compilation with build flags - the feature must always be compiled in, which conflicts with `BUILDFLAG(ENABLE_JXL_DECODER)`.

- **Feature**: `kJXLImageFormat` in `third_party/blink/public/common/features.h`
- **Status**: `base::FEATURE_ENABLED_BY_DEFAULT`
- **Control**: Can be disabled remotely via Finch without code push

Rollout Plan

Kill Switch

- **Fast**: Disable `kJXLImageFormat` via Finch experiment
- **Targeted**: Can disable per-platform, per-channel, or per-user cohort
- **Fallback**: Disable build flag in future release if needed

Metrics Monitored

- Usage: WebDX counter (`kJxl`)
- Performance: `Renderer4.ImageDecodeTaskDurationUs.Jxl`
- Memory: `Memory.Renderer.PrivateMemoryFootprint`
- Stability: Crash reports, fuzzer coverage

Core principle considerations

Speed

This change adds a new image decoder but does not modify existing code paths for other image formats. Performance impact is isolated to pages that serve JXL images.

Simplicity

No user-facing UI changes. This is a transparent capability addition. Users will simply see JXL images render correctly where they previously saw broken images or download prompts.

Security

CL includes a fuzzer.

Threat Model
Image decoders are a critical attack surface because they:

- Process untrusted data from the network
- Run in the renderer process
- Parse complex binary formats
- JPEG XL is a particularly complex format with entropy coding, color transforms, and animation support, all potential sources of vulnerabilities.

Mitigations:

- Memory Corruption -> jxl-rs uses Rust's memory safety for the bulk of its complex parsing logic, providing a significant reduction in attack surface compared to C++ alternatives. While the decoder strives for pure safety, it contains a minimal, tightly-scoped amount of unsafe code (both in the core crate and the jxl_simd crate) for performance-critical operations like SIMD intrinsics and low-level memory handling. These constrained blocks are subject to strict security review.
- C++ boundary issues -> base::span

Followup work

Implement progressive rendering, start by making sure upstream jxl-rs is supporting it. Plumb it down to the image decoder once it's in the jxl-rs. This PR is blocked right now.

Follow work on PRs: https://github.com/libjxl/jxl-rs/pulls/hjanuschka
The PR's there are optional, but nice to have and will improve chromium integration overall.