# Decoupling WebCodecs from Streams

**PUBLIC**

Dan Sanders <sandersd@chromium.org>, Chris Cunningham <chcunningham@chromium.org>

# Introduction

WebCodecs (https://github.com/WICG/web-codecs) is intended to be a low-level API for decoding and encoding media on the web.

The Streams API (https://streams.spec.whatwg.org/) seems to be a natural fit for codecs: codecs have a stream of input data, a stream of output data, and backpressure that propagates from output to input, exactly the same as a Streams Transformer.

We have found that integrating with Streams has required an inordinate amount of design time, distracting from our core goal. The designs we have tried did not hide the complexity from WebCodecs users.

This document describes our reasons for moving away from Streams in the design of WebCodecs.

# Challenges with Streams

## Our use of Streams is Complex

WebCodecs has several non-fatal control signals, including `configure`, `flush`, and `reset`.

A key consideration of any design using Streams is keeping the control and data planes in sync. For example, `configure` describes properties of the upcoming data, so it must have a clear position relative to data chunks. Existing Streams control signals are for managing stream lifecycle, so it is not obvious how to build on this foundation to offer WebCodecs control signals. We considered a number of designs, which can be grouped as described below.

### Everything In-Band

Data and control are always synchronized if control is done with special data chunks. `configure` and `flush` would each be a type of chunk describing the data chunks that follow them. This design is functional, but has a few drawbacks.

1. `reset` can't work this way: its purpose is to immediately drop ongoing work, so waiting in the queue is a non-starter. You could instead abort the codec's `WritableStream`, but

this adds work to re-establish the stream as the abort is fatal (such a design is explored below).

2. Only the downstream nodes learn that `configure` and `flush` succeed/fail/complete. In most codec applications both the app (conductor) and upstream nodes will need to know when these control operations complete. It's possible to create a backchannel from downstream nodes, but this violates the separation of responsibilities, distracting from the downstream node's core tasks (e.g. rendering, or muxing).

### Overloading existing control signals

In this design, `configure()` is added as a method on the Codec. `flush` and `reset` are implemented via the existing Streams `close()` and `abort()` algorithms, respectively.

Calling `configure()` would abort the `WritableStream` and return a promise. Upon completion, the promise resolves to provide a new `WritableStream`. Apps will generally want to `flush` prior to calling `configure()` to ensure that chunks of the old configuration aren't lost in internal queues upon the abort. Flushing the codec is performed by `close()`ing its `WritableStream`. Apps may want to initiate the `close()` upstream and have it propagate to ensure all nodes are flushed.

This design has a few issues:

1. Streams don't always indicate when `close()` completes. `WritableStream.close()` does indicate completion via promise, but `ReadableStream` is closed via `ReadableStreamDefaultController.close()`, which returns nothing.

2. It's complicated. There's a lot of Streams subtlety to understand here to perform basic codec operations.

## Lifecycle Mismatch

Both of the designs considered above rely on Stream's `abort` and `close` algorithms. Both are fatal to the stream and the fatality cascades to other nodes unless "prevented".

This is a burden for WebCodecs users. They will frequently need to reconstruct Streams and re-establish pipes for simple run-time operations like seeking and reconfiguration.

This is not intuitive for a codec API. These operations are not fatal to the codec itself and are not conceptually fatal to the binding of codec inputs and outputs.

## Streams are Difficult to Understand

Our proposed designs burden application developers with understanding all of the nuances of Streams to use WebCodecs. We know firsthand that this is not an easy task.

Streams users must constantly reason from both the internal and external perspective. This is especially tricky for implementers of transformers, which internally read from a `WritableStream` and write to a `ReadableStream`. WebCodecs users must implement at this level to correctly synchronize control signals as they (1) construct the graph and (2) implement their own streams for IO, or codec polyfills in WASM.

Once the graph is constructed, users must appreciate which control signals are available on which controllers, which of those cascade, in what directions, and their effect on (and synchronization with) the flow of chunks. By default, entering an error state for any node will cascade through the entire graph. Preventing the cascade with `preventAbort`/`preventClose` prevents cascading pipe closure, but it also makes the downstream oblivious that the upstream close has occurred.

After months of working with the Streams API, we are still often confused about the details above. MDN documentation is incomplete, and does not cover nuances of control signal propagation. We repeatedly need to reference the specification for class definitions and algorithm steps.

## A Different API may be Preferred

Existing codec APIs (including those in PPAPI) are class-like, so native apps are already written in that style. Ports of native apps are likely to wrap WebCodecs in a class-like interface. Given that our underlying implementation is also class-like, Streams appear to be a distraction.

With our prototypes, we found that trivial apps (demos) without significant control planes (user interactions, state changes, error handling, fallback behaviors, etc.) were naturally expressed with streams. Such apps are not expected to be the significant users of WebCodecs.

Even relatively simple uses, such as transcoding a uniform video stream, must handle potentially transient error cases, such as loss of graphics context. More complex cases, such as playback of a corrupt stream, require different handling in different apps (WebRTC can request a new keyframe from the sender, while VLC may want to fall back to a more resilient decoder).

# Future Integration with Streams

None of this prevents an app from wrapping WebCodecs in a Streams adapter, if that fits their use-case. We will try to design an API with an obvious mapping to Streams concepts (eg. provide `desiredSize()` for backpressure signaling).

We would consider standardizing a Streams-based API in the future if such a design gained adoption among WebCodecs users.