# Specification (draft)

> This is a draft. It is neither complete nor, probably, correct.

## Core concept

A Promistream is, at its core, simply a chain of Promises established through recursive asynchronous calls. A read is requested from the last stream in a pipeline (the "sink"), it requests a read from the stream upstream from it, which requests a read from the stream upstream to *it*, and so on - until the first stream in the pipeline, the "source", is reached.

Each stream along the way may choose to modify both the value itself, as well as the characteristics of it being produced; it may be delayed, filtered, turned into multiple values, or any other transformation that the project requires.

Therefore, the flow (simplified) is as follows:

```
USER CODE  --read-->  SINK STREAM  --read-->  TRANSFORM STREAM(S)  --read-->  SOURCE STREAM
--value-->  TRANSFORM STREAM(S) --value-->  SINK STREAM  --value-->  USER CODE
```

## Definitions

This is an overview of the terms used in this specification, as a reading aid; note that their descriptions here are limited, and **may not be sufficient to implement their concepts**. Further technical definition may exist elsewhere in the specification.

- **Stream**: An object which defines (at least) a function that reads a value from another stream and/or yields a value upon being read.
- **Pipeline**: A sequence of streams, which reads and their resulting values propagate through, potentially being transformed in the process.
- **Source stream**: The first stream in a pipeline. This is where values originally originate from. They may be generated on-the-fly, or originate from an external source of some kind.
- **Sink stream**: The last stream in a pipeline. This is where the read is originally initiated.
- **Transform stream**: A stream which exists in a pipeline somewhere inbetween a source and a sink stream.
- **Pipeline completion**: The point where an entire pipeline has been successfully read to its end; as defined by its source stream. A pipeline has only fully completed once all

internal buffers on all streams in the pipeline have run out.
- **Pipeline termination:** The point where a pipeline has been terminated or aborted prematurely. Again, it has only fully terminated once all internal buffers on all streams in the pipeline have run out.
- **Buffer:** A mechanism that any stream may have, where it internally stores some value(s) that are already known, but that it does not intend to yield or process until a next read. Frequently used to handle values that result in multiple transformed values (or vice versa), as well as format parsing.
- **Forking stream:** A stream which yields multiple other 'forks' (which are also streams), across which values are distributed or divided in some manner specific to the forking stream (mirroring, round-robin, etc.).
- **Converging stream:** The inverse of a forking stream; it takes multiple 'forks' and combines them back into one coherent stream, in some manner specific to the converging stream (combining, interspersing, etc.).
- **Upstream:** The stream(s) that come before the stream in question, ie. towards the direction of the source stream.
- **Downstream:** The stream(s) that come after the stream in question, ie. towards the direction of the sink stream.
- **EndOfStream marker:** A special error type that signals that the end of the (source) stream has been reached successfully. There will be no more data to read.
- **Aborted marker:** A special error type that signals that the pipeline has been terminated (ie. aborted) prematurely; either due to expected conditions, or due to an unexpected error.
- **Side-effects:** Any kind of externally observable change or interaction that a stream makes outside of its locally-defined state. This includes things like database queries, filesystem operations, but also eg. changing global variables or some other value that is 'owned' by something other than the stream.
- **Error:** Any object that inherits from the `Error` prototype/class, either directly or indirectly (eg. through a custom error type). Note the capitalization!

# Backpressure

TODO

# Stream API

A stream is any object which has the following properties:

**`promistreamVersion`:** Set to the value `0`, as of this version of the specification.

**`description`:** A short string containing a brief textual description (a few words at most) of the stream; this will typically be something like a package name, or a description of the purpose. This string is for debugging purposes only; it is not guaranteed or expected to be unique or in any particular format, but should be concise enough to be usable in a visualization of a pipeline.

**`abort`:** A function which, upon invocation, causes the stream to run its stream termination logic. Further defined below.

**`peek`:** A function which, upon invocation, determines the availability of more values upstream. Further defined below.

**`read`:** A function which, upon invocation, produces a value. Further defined below.

Whether something constitutes a valid stream, is determined solely by the object shape; the object does *not* need to inherit from any particular prototype, nor does it need to have any particular name.

Below, the behaviours of these functions will be detailed further; note that there will often be special technical requirements for different types of streams, that are detailed after the general description in a subsection.

# The `read` function

Signature: `read(source)`

This function should, upon invocation, produce a Promise which eventually resolves (to a read value) or rejects (with an error or marker). The stream MAY internally delay the settling of this Promise to control backpressure. The stream MAY (and typically will) consult the stream directly upstream from it (`source`), by calling its read function, in the process of producing its own value.

When the upstream read was successful, and a value was obtained, the produced Promise will resolve with that value. When the upstream read fails, the produced Promise will reject with an error or marker. Markers are simply Errors of a specific type, that signify the state of the source stream; the possible markers are detailed below.

When you read from upstream, you should be prepared to handle these markers if your stream is managing any resources that it needs to dispose of upon the pipeline completing or terminating (but it SHOULD only do so upon the first observation of a marker). Errors that are not markers can typically be left to propagate downstream; unless specified otherwise below. All markers and errors are rejections rather than resolutions, and so can be handled with a `catch` block, or will automatically propagate if left uncaught.

Note that the `read` function MAY be called while a previous `read` operation is still in flight (see "Ordering considerations" below), and streams should be prepared to handle this.

## Types of stream ending

There are three different ways in which a pipeline can end, signified by different markers:

- `EndOfStream`: The source stream reached its end successfully and all streams have drained their buffers. This is the success condition, and is initiated by the source stream.
- `Aborted` **(with** `true` **as reason)**: The source stream was prematurely terminated, but under expected conditions. This is typically initiated by code that is external to the pipeline, signifying that it does not need any further data from the pipeline, and the streams within it can dispose of any resources they are holding onto. This is arguably also a success condition.
- `Aborted` **(with an** `Error` **as reason)**: The source stream was prematurely terminated, due to an unexpected error. This may have been initiated by external code, but will typically happen from within a stream in the pipeline, when it encounters an unexpected failure. This is the failure condition.

# Dealing with uncertain reads

A thing to remember is that a `read` call MUST always produce a Promise which resolves to a value or rejects to an error or marker. It is not allowed (or possible) to respond with a "try again later"; in that case, your stream SHOULD return a Promise that will only be settled at that later moment.

This is especially important for eg. transform streams that do not have a 1:1 correspondence between the values it reads from upstream, and the values that it yields itself, such as filtering streams. In that case, the stream should typically implemented such that it continues reading from upstream until an acceptable value has been obtained, and only then yield that value.

# Ordering considerations

By default, streams SHOULD always process reads 'in order'; that is, if there is some kind of correspondence between the yielded values and the upstream reads, the order of these yielded values must also correspond, even in the face of multiple concurrent in-flight read operations. This is also true for markers and errors. Streams MAY produce values out of order, but if they do so, this MUST be clearly documented and serve a specific documented purpose.

If the stream's internal logic is incapable of processing concurrent requests, the stream MUST enforce sequential processing of inbound read requests through some sort of queueing mechanism.

-Non-normative- A stream may be composed with the off-the-shelf `sequentialize` stream to meet this requirement without implementing any custom queueing logic.

# Source streams

A source stream will have its `read` function invoked without a `source` argument, as there is no stream upstream of it, and source streams are expected to either generate their values or obtain them from some source external to the pipeline.

A source stream is responsible for producing (and rejecting with) markers when the pipeline has been completed or terminated. It may either reuse a previously-generated marker, or generate a

new one upon each `read` invocation.

## Sink streams

A sink stream MUST watch for non-marker rejections of upstream reads (ie. rejection with an Error), and if one is observed, call its upstream's `abort` function with that Error as an argument. This is critical to the correct propagation of automatic aborts for unexpected errors.

A sink stream MUST, upon the first post-buffer-drain encounter with an Aborted marker where the `reason` is an Error, reject with that Error (ie. *not* the marker). It MUST propagate the Aborted marker itself on subsequent reads. This ensures that calling code receives the original Error, in the same way that they would if streams had not been used.

A sink stream SHOULD, upon any encounter with an Aborted marker where the `reason` is `true`, reject with that marker to propagate it. It MAY choose to resolve with a value instead.

A sink stream MUST, upon the first read invocation, eventually either resolve with a value (even if it is `undefined`) or reject with an Error. It MAY respond to subsequent `read` invocations by propagating an `EndOfStream` or `Aborted` marker.

## Streams that buffer

If you are implementing a stream that has some sort of internal buffering, then there is a special consideration that you need to make for ended streams; when you either generate or receive an `EndOfStream`/`Aborted` marker, you should make sure to process your internal buffers prior to propagating it downstream. Often this will mean buffering up the marker internally, and only returning it once enough reads have occurred to exhaust the internal buffer (although you MAY implement any buffer-draining behaviour that is appropriate to the purpose of your stream).

Since `EndOfStream` and `Aborted` markers are final and reusable, you MAY buffer these up and continue yielding them on future reads in perpetuity, without invoking any further upstream reads, if this makes implementation easier for you.

# The `peek` function

Signature: `peek(source)`

This function should, upon invocation, produce a Promise which eventually resolves to a boolean, indicating whether more data is available at the source stream that has not been read yet. While this query MAY be answered by any stream in the pipeline if it has specific reason to need to do so, it SHOULD typically be propagated upstream (to `source`) as-is such that the source stream can answer it (unless specified otherwise below).

Note that any stream which answers `true` to a `peek` request MUST *reserve* or otherwise keep track of the 'peeked' items, to ensure that many subsequent `peek` requests will only result in as many

confirmations as there are actual items to be read. In the simplest implementation, this may simply be a counter of how far the pipeline has 'peeked ahead' (decremented by actual reads happening), but for more complex buffering situations it may be necessary to maintain buffers of actual values. A stream responding to a `peek` call MAY therefore invoke the upstream `read` function if necessary to implement its behaviour. however, it MAY NOT initiate any processing of the resulting value if doing so would cause side-effects.

The Promise that is produced for a `peek` call MAY be delayed by the stream, if this is necessary for determining an accurate answer to the query.

-Non-normative, rationale- The purpose of the `peek` function is to support safe parallelization, especially when unbounded; this ensures that even when a parallel reading implementation is allowed to have `Infinity` simultaneous reads, there will only ever be approximately as many in-flight reads as there are items readily available at the source stream. This prevents resource exhaustion. This cannot be implemented simply by doing many `read` calls, as a `read` may take a very long time to be processed and cause side-effects, thereby defeating the point - the `peek` call is instead meant to be propagated more or less directly to the source stream, bypassing any processing delays.

# The `abort` function

Signature: `abort(reason, source)`

This function should, upon invocation, do any teardown that is immediately needed when a pipeline is aborted, and then propagate the `abort` call to the stream upstream from it (`source`), passing on the `reason` as well. Note that this is not the only opportunity to do stream teardown; for teardown that is not immediately required, it will often be easier to handle this in the `read` implementation alongside `EndOfStream` markers, as an `Aborted` marker will be passed down upon the next read after the `abort` call has arrived at the source stream.

The `abort` function may be called with a `reason` of either `true` (to indicate a termination under expected conditions, eg. no further data is required) or an Error (to indicate a termination due to an unexpected error somewhere in the pipeline).

## Source streams

Source streams must additionally, upon invocation, set an internal 'aborted' flag. Subsequent reads (after any buffer draining) should produce `Aborted` markers.

# Sink streams

A sink stream is a stream which is expected to be placed at the end of a pipeline. Upon reading from it, it SHOULD start driving reads above it (eg. in an asynchronous loop), and eventually yield a value that represents the outcome of that read process. It MAY choose to support being read from

multiple times, if necessary to make it work ergonomically. The value it yields MAY be something that represents an in-progress read loop (such as a generator), rather than a conclusive value.

-Non-normative- Some examples of the values that a sink stream may yield:

- An array containing all of the values read from upstream, yielded when the pipeline completes. (collect)
- A stream in some other format, that produces values as they are read from upstream (to-node-stream)
- The last observed value, yielded when the pipeline completes (default behaviour of simple-sink)

---

Revision #1
Created 2024-12-11 12:38:39 UTC by joepie91
Updated 2024-12-11 12:41:33 UTC by joepie91