Streams

Living Standard — Last Updated 1 April 2019

Participate:

GitHub whatwg/streams (new issue, open issues)
IRC: #whatwg on Freenode

Commits:

GitHub whatwg/streams/commits
Snapshot as of this commit
@streamsstandard

Tests:

web-platform-tests streams/ (ongoing work)

Translations (non-normative):

日本語

Demos:

streams.spec.whatwg.org/demos

File an issue about the selected text

Abstract

This specification provides APIs for creating, composing, and consuming streams of data that map efficiently to low-level I/O primitives.

Table of Contents

File an issue about the selected text

File an issue about the selected text reamController (*x*)

File an issue about the selected text efaultControllerClearAlgorithms (*controller*)

File an issue about the selected text

1. Introduction §

This section is non-normative.

Large swathes of the web platform are built on streaming data: that is, data that is created, processed, and consumed in an incremental fashion, without ever reading all of it into memory. The Streams Standard provides a common set of APIs for creating and interfacing with such streaming data, embodied in readable streams, writable streams, and transform streams.

These APIs have been designed to efficiently map to low-level I/O primitives, including specializations for byte streams where appropriate. They allow easy composition of multiple streams into pipe chains, or can be used directly via readers and writers. Finally, they are designed to automatically provide backpressure and queuing.

This standard provides the base stream primitives which other parts of the web platform can use to expose their streaming data. For example, [FETCH] exposes `Response` bodies as `ReadableStream` instances. More generally, the platform is full of streaming abstractions waiting to be expressed as streams: multimedia streams, file streams, inter-global communication, and more benefit from being able to process data incrementally instead of buffering it all into memory and processing it in one go. By providing the foundation for these streams to be exposed to developers, the Streams Standard enables use cases like:

- Video effects: piping a readable video stream through a transform stream that applies effects in real time.
- Decompression: piping a file stream through a transform stream that selectively decompresses files from a `.tgz` archive, turning them into `img` elements as the user scrolls through an image gallery.
- Image decoding: piping an HTTP response stream through a transform stream that decodes bytes into bitmap data, and then through another transform that translates bitmaps into PNGs. If installed inside the `fetch` hook of a service worker, this would allow developers to transparently polyfill new image formats. [SERVICE-WORKERS]

Web developers can also use the APIs described here to create their own streams, with the same APIs as those provided by the platform. Other developers can then transparently compose platform-provided streams with those supplied by libraries. In this way, the APIs described here provide unifying abstraction for all streams, encouraging an ecosystem to grow around these shared and composable interfaces.

2. Model §

A **chunk** is a single piece of data that is written to or read from a stream. It can be of any type; streams can even contain chunks of different types. A chunk will often not be the most atomic unit of data for a given stream; for example a byte stream might contain chunks consisting of 16 KiB `Uint8Array`s, instead of single bytes.

2.1. Readable streams §

A **readable stream** represents a source of data, from which you can read. In other words, data comes *out* of a readable stream. Concretely, a readable stream is an instance of the `ReadableStream` class.

Although a readable stream can be created with arbitrary behavior, most readable streams wrap a lower-level I/O source, called the **underlying source**. There are two types of underlying source: push sources and pull sources.

Push sources push data at you, whether or not you are listening for it. They may also provide a mechanism for pausing and resuming the flow of data. An example push source is a TCP socket, where data is constantly being pushed from the OS level, at a rate that can be controlled by changing the TCP window size.

Pull sources require you to request data from them. The data may be available synchronously, e.g. if it is held by the operating system's in-memory buffers, or asynchronously, e.g. if it has to be read from disk. An example pull source is a file handle, where you seek to specific locations and read specific amounts.

Readable streams are designed to wrap both types of sources behind a single, unified interface. For web developer–created streams, the implementation details of a source are provided by an object with certain methods and properties that is passed to the `ReadableStream()` constructor.

Chunks are enqueued into the stream by the stream's underlying source. They can then be read one at a time via the stream's public interface, in particular by using a readable stream reader acquired using the stream's `getReader()` method.

Code that reads from a readable stream using its public interface is known as a **consumer**.

Consumers also have the ability to **cancel** a readable stream, using its `cancel()` method. This indicates that the consumer has lost interest in the stream, and will immediately close the stream, throw away any queued chunks, and execute any cancellation mechanism of the underlying source.

Consumers can also **tee** a readable stream using its `tee()` method. This will lock the stream, making it no longer directly usable; however, it will create two new streams, called **branches**, which can be consumed independently.

For streams representing bytes, an extended version of the readable stream is provided to handle bytes efficiently, in particular by minimizing copies. The underlying source for such a readable stream is called an **underlying byte source**. A readable stream whose underlying source is an underlying byte source is sometimes called a **readable byte stream**. Consumers of a readable byte stream can acquire a BYOB reader using the stream's `getReader()` method.

2.2. Writable streams §

A **writable stream** represents a destination for data, into which you can write. In other words, data goes *in* to a writable stream. Concretely, a writable stream is an instance of the `WritableStream` class.

Analogously to readable streams, most writable streams wrap a lower-level I/O sink, called the **underlying sink**. Writable streams work to abstract away some of the complexity of the underlying sink, by queuing subsequent writes and only delivering them to the underlying sink one by one.

Chunks are written to the stream via its public interface, and are passed one at a time to the stream's underlying sink. For web developer-created streams, the implementation details of the sink are provided by an object with certain methods that is passed to the `WritableStream()` constructor.

Code that writes into a writable stream using its public interface is known as a **producer**.

Producers also have the ability to **abort** a writable stream, using its `abort()` method. This indicates that the producer believes something has gone wrong, and that future writes should be discontinued. It puts the stream in an errored state, even without a signal from the underlying sink, and it discards all writes in the stream's internal queue.

2.3. Transform streams §

A **transform stream** consists of a pair of streams: a writable stream, known as its **writable side**, and a readable stream, known as its **readable side**. In a manner specific to the transform stream in question, writes to the writable side result in new data being made available for reading from the readable side.

Concretely, any object with a `writable` property and a `readable` property can serve as a transform stream. However, the standard `TransformStream` class makes it much easier to create such a pair that is properly entangled. It wraps a **transformer**, which defines algorithms for the specific transformation to be performed. For web developer–created streams, the implementation details of a transformer are provided by an object with certain methods and properties that is passed to the `TransformStream()` constructor.

An **identity transform stream** is a type of transform stream which forwards all chunks written to its writable side to its readable side, without any changes. This can be useful in a variety of scenarios. By default, the `TransformStream` constructor will create an identity transform stream, when no `transform()` method is present on the transformer object.

Some examples of potential transform streams include:

- A GZIP compressor, to which uncompressed bytes are written and from which compressed bytes are read;
- A video decoder, to which encoded bytes are written and from which uncompressed video frames are read;
- A text decoder, to which bytes are written and from which strings are read;
- A CSV-to-JSON converter, to which strings representing lines of a CSV file are written and from which corresponding JavaScript objects are read.

2.4. Pipe chains and backpressure §

Streams are primarily used by **piping** them to each other. A readable stream can be piped directly to a writable stream, using its `pipeTo()` method, or it can be piped through one or more transform streams first, using its `pipeThrough()` method.

A set of streams piped together in this way is referred to as a **pipe chain**. In a pipe chain, the **original source** is the underlying source of the first readable stream in the chain; the **ultimate sink** is the underlying sink of the final writable stream in the chain.

Once a pipe chain is constructed, it will propagate signals regarding how fast chunks should flow through it. If any step in the chain cannot yet accept chunks, it propagates a signal backwards through the pipe chain, until eventually the original source is told to stop producing chunks so fast. This process of normalizing flow from the original source according to how fast the chain can process chunks is called **backpressure**.

Concretely, the original source is given the `controller.desiredSize` (or `byteController.desiredSize`) value, and can then adjust its rate of data flow accordingly. This value is derived from the `writer.desiredSize` corresponding to the ultimate sink, which gets updated as the ultimate sink finishes writing chunks. The `pipeTo()` method used to construct the chain automatically ensures this information propagates back through the pipe chain.

When teeing a readable stream, the backpressure signals from its two branches will aggregate, such that if neither branch is read from, a backpressure signal will be sent to the underlying source of the original stream.

Piping locks the readable and writable streams, preventing them from being manipulated for the duration of the pipe operation. This allows the implementation to perform important optimizations, such as directly shuttling data from the underlying source to the underlying sink while bypassing many of the intermediate queues.

2.5. Internal queues and queuing strategies §

Both readable and writable streams maintain **internal queues**, which they use for similar purposes. In the case of a readable stream, the internal queue contains chunks that have been enqueued by the underlying source, but not yet read by the consumer. In the case of a writable stream, the internal queue contains chunks which have been written to the stream by the producer, but not yet processed and acknowledged by the underlying sink.

A **queuing strategy** is an object that determines how a stream should signal backpressure based on the state of its internal queue. The queuing strategy assigns a size to each chunk, and compares the total size of all chunks in the queue to a specified number, known as the **high water mark**. The resulting difference, high water mark minus total size, is used to determine the **desired size to fill the stream's queue**.

For readable streams, an underlying source can use this desired size as a backpressure signal, slowing down chunk generation so as to try to keep the desired size above or at zero. For writable streams, a producer can behave similarly, avoiding writes that would cause the desired size to go negative.

Concretely, a queuing strategy for web developer–created streams is given by any JavaScript object with a `highWaterMark` property. For byte streams the `highWaterMark` always has units of bytes. For other streams the default unit is chunks, but a `size()` function can be included in the

strategy object which returns the size for a given chunk. This permits the `highWaterMark` to be specified in arbitrary floating-point units.

Example

A simple example of a queuing strategy would be one that assigns a size of one to each chunk, and has a high water mark of three. This would mean that up to three chunks could be enqueued in a readable stream, or three chunks written to a writable stream, before the streams are considered to be applying backpressure.

In JavaScript, such a strategy could be written manually as `{ highWaterMark: 3, size() { return 1; }}`, or using the built-in `CountQueuingStrategy` class, as `new CountQueuingStrategy({ highWaterMark: 3 })`.

2.6. Locking §

A **readable stream reader**, or simply reader, is an object that allows direct reading of chunks from a readable stream. Without a reader, a consumer can only perform high-level operations on the readable stream: canceling the stream, or piping the readable stream to a writable stream. A reader is acquired via the stream's `getReader()` method.

A readable byte stream has the ability to vend two types of readers: **default readers** and **BYOB readers**. BYOB ("bring your own buffer") readers allow reading into a developer-supplied buffer, thus minimizing copies. A non-byte readable stream can only vend default readers. Default readers are instances of the `ReadableStreamDefaultReader` class, while BYOB readers are instances of `ReadableStreamBYOBReader`.

Similarly, a **writable stream writer**, or simply writer, is an object that allows direct writing of chunks to a writable stream. Without a writer, a producer can only perform the high-level operations of aborting the stream or piping a readable stream to the writable stream. Writers are represented by the `WritableStreamDefaultWriter` class.

Note

Under the covers, these high-level operations actually use a reader or writer themselves.

A given readable or writable stream only has at most one reader or writer at a time. We say in this case the stream is **locked**, and that the reader or writer is **active**. This state can be determined using the `readableStream.locked` or `writableStream.locked` properties.

A reader or writer also has the capability to **release its lock**, which makes it no longer active, and allows further readers or writers to be acquired. This is done via the `defaultReader.releaseLock()`, `byobReader.releaseLock()`, or `writer.releaseLock()` method, as appropriate.

3. Readable streams §

3.1. Using readable streams §

Example

The simplest way to consume a readable stream is to simply pipe it to a writable stream. This ensures that backpressure is respected, and any errors (either writing or reading) are propagated through the chain:

```
readableStream.pipeTo(writableStream)
  .then(() => console.log("All data successfully written!"))
  .catch(e => console.error("Something went wrong!", e));
```

Example

If you simply want to be alerted of each new chunk from a readable stream, you can pipe it to a new writable stream that you custom-create for that purpose:

```
readableStream.pipeTo(new WritableStream({
  write(chunk) {
    console.log("Chunk received", chunk);
  },
  close() {
    console.log("All data successfully read!");
  },
  abort(e) {
    console.error("Something went wrong!", e);
  }
}));
```

By returning promises from your `write()` implementation, you can signal backpressure to the readable stream.

Example

Although readable streams will usually be used by piping them to a writable stream, you can also read them directly by acquiring a reader and using its `read()` method to get successive chunks. For example, this code logs the next chunk in the stream, if available:

```
const reader = readableStream.getReader();

reader.read().then(
  ({ value, done }) => {
    if (done) {
      console.log("The stream was already closed!");
    } else {
      console.log(value);
    }
  },
  e => console.error("The stream became errored and cannot be read from!", e)
);
```

This more manual method of reading a stream is mainly useful for library authors building new high-level operations on streams, beyond the provided ones of piping and teeing.

Example

The above example showed using the readable stream's default reader. If the stream is a readable byte stream, you can also acquire a BYOB reader for it, which allows more precise control over buffer allocation in order to avoid copies. For example, this code reads the first 1024 bytes from the stream into a single memory buffer:

```
const reader = readableStream.getReader({ mode: "byob" });

let startingAB = new ArrayBuffer(1024);
readInto(startingAB)
  .then(buffer => console.log("The first 1024 bytes:", buffer))
  .catch(e => console.error("Something went wrong!", e));
```

File an issue about the selected text

```
function readInto(buffer, offset = 0) {
  if (offset === buffer.byteLength) {
    return Promise.resolve(buffer);
  }

  const view = new Uint8Array(buffer, offset, buffer.byteLength - offset);
  return reader.read(view).then(newView => {
    return readInto(newView.buffer, offset + newView.byteLength);
  });
}
```

An important thing to note here is that the final `buffer` value is different from the `startingAB`, but it (and all intermediate buffers) shares the same backing memory allocation. At each step, the buffer is transferred to a new `ArrayBuffer` object. The `newView` is a new `Uint8Array`, with that `ArrayBuffer` object as its `buffer` property, the offset that bytes were written to as its `byteOffset` property, and the number of bytes that were written as its `byteLength` property.

3.2. Class `ReadableStream`

The `ReadableStream` class is a concrete instance of the general readable stream concept. It is adaptable to any chunk type, and maintains an internal queue to keep track of data supplied by the underlying source but not yet read by any consumer.

3.2.1. Class definition §

This section is non-normative.

If one were to write the `ReadableStream` class in something close to the syntax of [ECMASCRIPT], it would look like

```
class ReadableStream {
  constructor(underlyingSource = {}, strategy = {})

  get locked()

  cancel(reason)
  getIterator({ preventCancel } = {})
  getReader({ mode } = {})
  pipeThrough({ writable, readable },
              { preventClose, preventAbort, preventCancel, signal } = {})
  pipeTo(dest, { preventClose, preventAbort, preventCancel, signal } = {})
  tee()

  [@@asyncIterator]({ preventCancel } = {})
}
```

3.2.2. Internal slots §

Instances of `ReadableStream` are created with the internal slots described in the following table:

Internal Slot	Description (*non-normative*)
[[disturbed]]	A boolean flag set to **true** when the stream has been read from or canceled
[[readableStreamController]]	A `ReadableStreamDefaultController` or `ReadableByteStreamController` created with the ability to control the state and queue of this stream; also used for the IsReadableStream brand check
[[reader]]	A `ReadableStreamDefaultReader` or `ReadableStreamBYOBReader` instance, if the stream is locked to a reader, or **undefined** if it is not
[[state]]	A string containing the stream's current state, used internally; one of `"readable"`, `"closed"`, or `"errored"`
[[storedError]]	A value indicating how the stream failed, to be given as a failure reason or exception when trying to operate on an errored stream

3.2.3. new ReadableStream(*underlyingSource* = {}, *strategy* = {})

File an issue about the selected text

Note: *The `underlyingSource` argument represents the underlying source, as described in §3.2.4 Underlying source API.*

The `strategy` argument represents the stream's queuing strategy, as described in §6.1.1 The queuing strategy API. If it is not provided, the default behavior will be the same as a `CountQueuingStrategy` with a high water mark of 1.

1. Perform ! InitializeReadableStream(**this**).
2. Let *size* be ? GetV(*strategy*, `"size"`).
3. Let *highWaterMark* be ? GetV(*strategy*, `"highWaterMark"`).
4. Let *type* be ? GetV(*underlyingSource*, `"type"`).
5. Let *typeString* be ? ToString(*type*).
6. If *typeString* is `"bytes"`,
    a. If *size* is not **undefined**, throw a **RangeError** exception.
    b. If *highWaterMark* is **undefined**, let *highWaterMark* be **0**.
    c. Set *highWaterMark* to ? ValidateAndNormalizeHighWaterMark(*highWaterMark*).
    d. Perform ? SetUpReadableByteStreamControllerFromUnderlyingSource(**this**, *underlyingSource*, *highWaterMark*).
7. Otherwise, if *type* is **undefined**,
    a. Let *sizeAlgorithm* be ? MakeSizeAlgorithmFromSizeFunction(*size*).
    b. If *highWaterMark* is **undefined**, let *highWaterMark* be **1**.
    c. Set *highWaterMark* to ? ValidateAndNormalizeHighWaterMark(*highWaterMark*).
    d. Perform ? SetUpReadableStreamDefaultControllerFromUnderlyingSource(**this**, *underlyingSource*, *highWaterMark*, *sizeAlgorithm*).
8. Otherwise, throw a **RangeError** exception.

3.2.4. Underlying source API §

This section is non-normative.

The `ReadableStream()` constructor accepts as its first argument a JavaScript object representing the underlying source. Such objects can contain any of the following properties:

`start(controller)`

A function that is called immediately during creation of the `ReadableStream`.

Typically this is used adapt a push source by setting up relevant event listeners, as in the example of §8.1 A readable stream with an underlying push source (no backpressure support), or to acquire access to a pull source, as in §8.4 A readable stream with an underlying pull source.

If this setup process is asynchronous, it can return a promise to signal success or failure; a rejected promise will error the stream. Any thrown exceptions will be re-thrown by the `ReadableStream()` constructor.

`pull(controller)`

A function that is called whenever the stream's internal queue of chunks becomes not full, i.e. whenever the queue's desired size becomes positive. Generally, it will be called repeatedly until the queue reaches its high water mark (i.e. until the desired size becomes non-positive).

For push sources, this can be used to resume a paused flow, as in §8.2 A readable stream with an underlying push source and backpressure support. For pull sources, it is used to acquire new chunks to enqueue into the stream, as in §8.4 A readable stream with an underlying pull source.

This function will not be called until `start()` successfully completes. Additionally, it will only be called repeatedly if it enqueues at least one chunk or fulfills a BYOB request; a no-op `pull()` implementation will not be continually called.

If the function returns a promise, then it will not be called again until that promise fulfills. (If the promise rejects, the stream will become errored.) This is mainly used in the case of pull sources, where the promise returned represents the process of acquiring a new chunk. Throwing an exception is treated the same as returning a rejected promise.

`cancel(reason)`

A function that is called whenever the consumer cancels the stream, via `stream.cancel()`, `defaultReader.cancel()`, or `byobReader.cancel()`. It takes as its argument the same value as was passed to those methods by the consumer.

Readable streams can additionally be canceled under certain conditions during piping; see the definition of the `pipeTo()` method for more details.

For all streams, this is generally used to release access to the underlying resource; see for example §8.1 A readable stream with an underlying push source (no backpressure support).

If the shutdown process is asynchronous, it can return a promise to signal success or failure; the result will be communicated via the return value of the `cancel()` method that was called. Additionally, a rejected promise will error the stream, instead of letting it close. Throwing an exception is treated the same as returning a rejected promise.

`type` (byte streams only)

Can be set to `"bytes"` to signal that the constructed `ReadableStream` is a readable byte stream. This ensures that the resulting `ReadableStream` will successfully be able to vend BYOB readers via its `getReader()` method. It also affects the `controller` argument passed to the `start()` and `pull()` methods; see below.

For an example of how to set up a readable byte stream, including using the different controller interface, see §8.3 A readable byte stream with an underlying push source (no backpressure support).

Setting any value other than `"bytes"` or **undefined** will cause the `ReadableStream()` constructor to throw an exception.

autoAllocateChunkSize **(byte streams only)**

Can be set to a positive integer to cause the implementation to automatically allocate buffers for the underlying source code to write into. In this case, when a consumer is using a default reader, the stream implementation will automatically allocate an `ArrayBuffer` of the given size, so that `controller.byobRequest` is always present, as if the consumer was using a BYOB reader.

This is generally used to cut down on the amount of code needed to handle consumers that use default readers, as can be seen by comparing §8.3 A readable byte stream with an underlying push source (no backpressure support) without auto-allocation to §8.5 A readable byte stream with an underlying pull source with auto-allocation.

The type of the `controller` argument passed to the `start()` and `pull()` methods depends on the value of the `type` option. If `type` is set to `undefined` (including via omission), `controller` will be a `ReadableStreamDefaultController`. If it's set to `"bytes"`, `controller` will be a `ReadableByteStreamController`.

3.2.5. Properties of the `ReadableStream` prototype §

3.2.5.1. get locked

Note

The `locked` getter returns whether or not the readable stream is locked to a reader.

1. If ! IsReadableStream(**this**) is **false**, throw a **TypeError** exception.
2. Return ! IsReadableStreamLocked(**this**).

3.2.5.2. cancel(*reason*)

Note

The `cancel` method cancels the stream, signaling a loss of interest in the stream by a consumer. The supplied `reason` argument will be given to the underlying source's `cancel()` method, which might or might not use it.

1. If ! IsReadableStream(**this**) is **false**, return a promise rejected with a **TypeError** exception.
2. If ! IsReadableStreamLocked(**this**) is **true**, return a promise rejected with a **TypeError** exception.
3. Return ! ReadableStreamCancel(**this**, *reason*).

3.2.5.3. getIterator({ *preventCancel* } = {})

Note

The `getIterator` method returns an async iterator which can be used to consume the stream. The `return()` method of this iterator object will, by default, cancel the stream; it will also release the reader.

1. If ! IsReadableStream(**this**) is **false**, throw a **TypeError** exception.
2. Let *reader* be ? AcquireReadableStreamDefaultReader(**this**).
3. Let *iterator* be ! ObjectCreate(`ReadableStreamAsyncIteratorPrototype`).
4. Set *iterator*.[[asyncIteratorReader]] to *reader*.
5. Set *iterator*.[[preventCancel]] to ! ToBoolean(*preventCancel*).
6. Return *iterator*.

3.2.5.4. getReader({ *mode* } = {})

Note

The `getReader` method creates a reader of the type specified by the `mode` option and locks the stream to the new reader. While the stream is locked, no other reader can be acquired until this one is released.

This functionality is especially useful for creating abstractions that desire the ability to consume a stream in its entirety. By getting a reader for the stream, you can ensure nobody else can interleave reads with yours or cancel the stream, which would interfere with your abstraction.

*When `mode` is **undefined**, the method creates a default reader (an instance of `ReadableStreamDefaultReader`). The reader provides the ability to directly read individual chunks from the stream via the reader's `read()` method.*

When `mode` is `"byob"`, the `getReader` method creates a BYOB reader (an instance of `ReadableStreamBYOBReader`). This feature only works on readable byte streams, i.e. streams which were constructed specifically with the ability to handle "bring your own buffer" reading. The reader provides the ability to directly read individual chunks from the stream via the reader's `read()` method, into developer-supplied buffers, allowing more precise control over allocation.

1. If ! IsReadableStream(**this**) is **false**, throw a **TypeError** exception.
2. If *mode* is **undefined**, return ? AcquireReadableStreamDefaultReader(**this**, **true**).
3. Set *mode* to ? ToString(*mode*).
4. If *mode* is `"byob"`, return ? AcquireReadableStreamBYOBReader(**this**, **true**).
5. Throw a **RangeError** exception.

Example

An example of an abstraction that might benefit from using a reader is a function like the following, which is designed to read an entire readable stream into memory as an array of chunks.

```
function readAllChunks(readableStream) {
  const reader = readableStream.getReader();
  const chunks = [];

  return pump();

  function pump() {
    return reader.read().then(({ value, done }) => {
      if (done) {
        return chunks;
      }

      chunks.push(value);
      return pump();
    });
  }
}
```

Note how the first thing it does is obtain a reader, and from then on it uses the reader exclusively. This ensures that no other consumer can interfere with the stream, either by reading chunks or by canceling the stream.

3.2.5.5. pipeThrough({ *writable*, *readable* }, { *preventClose*, *preventAbort*, *preventCancel*, *signal* } = {})

Note

The `pipeThrough` method provides a convenient, chainable way of piping this readable stream through a transform stream (or any other `{ writable, readable }` pair). It simply pipes the stream into the writable side of the supplied pair, and returns the readable side for further use.

Piping a stream will lock it for the duration of the pipe, preventing any other consumer from acquiring a reader.

1. If ! IsReadableStream(**this**) is **false**, throw a **TypeError** exception.
2. If ! IsWritableStream(*writable*) is **false**, throw a **TypeError** exception.
3. If ! IsReadableStream(*readable*) is **false**, throw a **TypeError** exception.
4. Set *preventClose* to ! ToBoolean(*preventClose*), set *preventAbort* to ! ToBoolean(*preventAbort*), and set *preventCancel* to ! ToBoolean(*preventCancel*).
5. If *signal* is not **undefined**, and *signal* is not an instance of the `AbortSignal` interface, throw a **TypeError** exception.
6. If ! IsReadableStreamLocked(**this**) is **true**, throw a **TypeError** exception.
7. If ! IsWritableStreamLocked(*writable*) is **true**, throw a **TypeError** exception.
8. Let *promise* be ! ReadableStreamPipeTo(**this**, *writable*, *preventClose*, *preventAbort*, *preventCancel*, *signal*).
9. Set *promise*.[[PromiseIsHandled]] to **true**.
10. Return *readable*.

Example

A typical example of constructing pipe chain using `pipeThrough(transform, options)` would look like

File an issue about the selected text

```
httpResponseBody
  .pipeThrough(decompressorTransform)
  .pipeThrough(ignoreNonImageFilesTransform)
  .pipeTo(mediaGallery);
```

3.2.5.6. pipeTo(*dest*, { *preventClose*, *preventAbort*, *preventCancel*, *signal* } = {})

Note

The `pipeTo` method pipes this readable stream to a given writable stream. The way in which the piping process behaves under various error conditions can be customized with a number of passed options. It returns a promise that fulfills when the piping process completes successfully, or rejects if any errors were encountered.

Piping a stream will lock it for the duration of the pipe, preventing any other consumer from acquiring a reader.

Errors and closures of the source and destination streams propagate as follows:

- *An error in the source readable stream will abort the destination writable stream, unless `preventAbort` is truthy. The returned promise will be rejected with the source's error, or with any error that occurs during aborting the destination.*
- *An error in the destination writable stream will cancel the source readable stream, unless `preventCancel` is truthy. The returned promise will be rejected with the destination's error, or with any error that occurs during canceling the source.*
- *When the source readable stream closes, the destination writable stream will be closed, unless `preventClose` is true. The returned promise will be fulfilled once this process completes, unless an error is encountered while closing the destination, in which case it will be rejected with that error.*
- *If the destination writable stream starts out closed or closing, the source readable stream will be canceled, unless `preventCancel` is true. The returned promise will be rejected with an error indicating piping to a closed stream failed, or with any error that occurs during canceling the source.*

The `signal` option can be set to an `AbortSignal` to allow aborting an ongoing pipe operation via the corresponding `AbortController`. In this case, the source readable stream will be canceled, and the destination writable stream aborted, unless the respective options `preventCancel` or `preventAbort` are set.

1. If ! IsReadableStream(**this**) is **false**, return a promise rejected with a **TypeError** exception.
2. If ! IsWritableStream(*dest*) is **false**, return a promise rejected with a **TypeError** exception.
3. Set *preventClose* to ! ToBoolean(*preventClose*), set *preventAbort* to ! ToBoolean(*preventAbort*), and set *preventCancel* to ! ToBoolean(*preventCancel*).
4. If *signal* is not **undefined**, and *signal* is not an instance of the `AbortSignal` interface, return a promise rejected with a **TypeError** exception.
5. If ! IsReadableStreamLocked(**this**) is **true**, return a promise rejected with a **TypeError** exception.
6. If ! IsWritableStreamLocked(*dest*) is **true**, return a promise rejected with a **TypeError** exception.
7. Return ! ReadableStreamPipeTo(**this**, *dest*, *preventClose*, *preventAbort*, *preventCancel*, *signal*).

3.2.5.7. tee()

Note

The `tee` method tees this readable stream, returning a two-element array containing the two resulting branches as new `ReadableStream` instances.

Teeing a stream will lock it, preventing any other consumer from acquiring a reader. To cancel the stream, cancel both of the resulting branches; a composite cancellation reason will then be propagated to the stream's underlying source.

Note that the chunks seen in each branch will be the same object. If the chunks are not immutable, this could allow interference between the two branches.

1. If ! IsReadableStream(**this**) is **false**, throw a **TypeError** exception.
2. Let *branches* be ? ReadableStreamTee(**this**, **false**).
3. Return ! CreateArrayFromList(*branches*).

Example

Teeing a stream is most useful when you wish to let two independent consumers read from the stream in parallel, perhaps even at different speeds. For example, given a writable stream `cacheEntry` representing an on-disk file, and another writable stream `httpRequestBody` representing an upload to a remote server, you could pipe the same readable stream to both destinations at once:

File an issue about the selected text

```
const [forLocal, forRemote] = readableStream.tee();

Promise.all([
  forLocal.pipeTo(cacheEntry),
  forRemote.pipeTo(httpRequestBody)
])
.then(() => console.log("Saved the stream to the cache and also uploaded it!"))
.catch(e => console.error("Either caching or uploading failed: ", e));
```

3.2.5.8. [@@asyncIterator]({ *preventCancel* } = {})

Note

The `@@asyncIterator` method is an alias of `getIterator()`.

The initial value of the `@@asyncIterator` method is the same function object as the initial value of the `getIterator()` method.

3.3. ReadableStreamAsyncIteratorPrototype

`ReadableStreamAsyncIteratorPrototype` is an ordinary object that is used by `getIterator()` to construct the objects it returns. Instances of `ReadableStreamAsyncIteratorPrototype` implement the `AsyncIterator` abstract interface from the JavaScript specification. [ECMASCRIPT]

The `ReadableStreamAsyncIteratorPrototype` object must have its [[Prototype]] internal slot set to `%AsyncIteratorPrototype%`.

3.3.1. Internal slots §

Objects created by `getIterator()`, using `ReadableStreamAsyncIteratorPrototype` as their prototype, are created with the internal slots described in the following table:

Internal Slot	Description (*non-normative*)
[[asyncIteratorReader]]	A `ReadableStreamDefaultReader` instance
[[preventCancel]]	A boolean value indicating if the stream will be canceled when the async iterator's `return()` method is called

3.3.2. next() §

1. If ! IsReadableStreamAsyncIterator(**this**) is **false**, return a promise rejected with a **TypeError** exception.
2. Let *reader* be **this**.[[asyncIteratorReader]].
3. If *reader*.[[ownerReadableStream]] is **undefined**, return a promise rejected with a **TypeError** exception.
4. Return the result of transforming ! ReadableStreamDefaultReaderRead(*reader*) with a fulfillment handler which takes the argument *result* and performs the following steps:
    a. Assert: Type(*result*) is Object.
    b. Let *done* be ! Get(*result*, `"done"`).
    c. Assert: Type(*done*) is Boolean.
    d. If *done* is **true**, perform ! ReadableStreamReaderGenericRelease(*reader*).
    e. Let *value* be ! Get(*result*, `"value"`).
    f. Return ! ReadableStreamCreateReadResult(*value*, *done*, **true**).

3.3.3. return(*value*)

1. If ! IsReadableStreamAsyncIterator(**this**) is **false**, return a promise rejected with a **TypeError** exception.
2. Let *reader* be **this**.[[asyncIteratorReader]].
3. If *reader*.[[ownerReadableStream]] is **undefined**, return a promise rejected with a **TypeError** exception.
4. If *reader*.[[readRequests]] is not empty, return a promise rejected with a **TypeError** exception.
5. If **this**.[[preventCancel]] is **false**, then:
    a. Let *result* be ! ReadableStreamReaderGenericCancel(*reader*, *value*).
    b. Perform ! ReadableStreamReaderGenericRelease(*reader*).
    c. Return the result of transforming *result* by a fulfillment handler that returns ! ReadableStreamCreateReadResult(*value*, **true**, **true**).

File an issue about the selected text ıamReaderGenericRelease(*reader*).

7. Return a promise resolved with ! ReadableStreamCreateReadResult(*value*, **true**, **true**).

3.4. General readable stream abstract operations §

The following abstract operations, unlike most in this specification, are meant to be generally useful by other specifications, instead of just being part of the implementation of this spec's classes.

3.4.1. AcquireReadableStreamBYOBReader (*stream*[, *forAuthorCode*]) throws §

This abstract operation is meant to be called from other specifications that may wish to acquire a BYOB reader for a given stream.

1. If *forAuthorCode* was not passed, set it to **false**.
2. Let *reader* be ? Construct(`ReadableStreamBYOBReader`, « *stream* »).
3. Set *reader*.[[forAuthorCode]] to *forAuthorCode*.
4. Return *reader*.

3.4.2. AcquireReadableStreamDefaultReader (*stream*[, *forAuthorCode*]) throws §

This abstract operation is meant to be called from other specifications that may wish to acquire a default reader for a given stream.

Note

Other specifications ought to leave forAuthorCode *as its default value of* **false**, *unless they are planning to directly expose the resulting* `{ value, done }` *object to authors. See the note regarding ReadableStreamCreateReadResult for more information.*

1. If *forAuthorCode* was not passed, set it to **false**.
2. Let *reader* be ? Construct(`ReadableStreamDefaultReader`, « *stream* »).
3. Set *reader*.[[forAuthorCode]] to *forAuthorCode*.
4. Return *reader*.

3.4.3. CreateReadableStream (*startAlgorithm*, *pullAlgorithm*, *cancelAlgorithm* [, *highWaterMark* [, *sizeAlgorithm*]]) throws §

This abstract operation is meant to be called from other specifications that wish to create `ReadableStream` instances. The *pullAlgorithm* and *cancelAlgorithm* algorithms must return promises; if supplied, *sizeAlgorithm* must be an algorithm accepting chunk objects and returning a number; and if supplied, *highWaterMark* must be a non-negative, non-NaN number.

Note

CreateReadableStream throws an exception if and only if the supplied startAlgorithm *throws.*

1. If *highWaterMark* was not passed, set it to **1**.
2. If *sizeAlgorithm* was not passed, set it to an algorithm that returns **1**.
3. Assert: ! IsNonNegativeNumber(*highWaterMark*) is **true**.
4. Let *stream* be ObjectCreate(the original value of `ReadableStream`'s `prototype` property).
5. Perform ! InitializeReadableStream(*stream*).
6. Let *controller* be ObjectCreate(the original value of `ReadableStreamDefaultController`'s `prototype` property).
7. Perform ? SetUpReadableStreamDefaultController(*stream*, *controller*, *startAlgorithm*, *pullAlgorithm*, *cancelAlgorithm*, *highWaterMark*, *sizeAlgorithm*).
8. Return *stream*.

3.4.4. CreateReadableByteStream (*startAlgorithm*, *pullAlgorithm*, *cancelAlgorithm* [, *highWaterMark* [, *autoAllocateChunkSize*]]) throws §

This abstract operation is meant to be called from other specifications that wish to create `ReadableStream` instances of type "bytes". The *pullAlgorithm* and *cancelAlgorithm* algorithms must return promises; if supplied, *highWaterMark* must be a non-negative, non-NaN number, and if supplied, *autoAllocateChunkSize* must be a positive integer.

Note

CreateReadableByteStream throws an exception if and only if the supplied startAlgorithm *throws.*

File an issue about the selected text

1. If *highWaterMark* was not passed, set it to **0**.
2. If *autoAllocateChunkSize* was not passed, set it to **undefined**.
3. Assert: ! IsNonNegativeNumber(*highWaterMark*) is **true**.
4. If *autoAllocateChunkSize* is not **undefined**,
    a. Assert: ! IsInteger(*autoAllocateChunkSize*) is **true**.
    b. Assert: *autoAllocateChunkSize* is positive.
5. Let *stream* be ObjectCreate(the original value of `ReadableStream`'s `prototype` property).
6. Perform ! InitializeReadableStream(*stream*).
7. Let *controller* be ObjectCreate(the original value of `ReadableByteStreamController`'s `prototype` property).
8. Perform ? SetUpReadableByteStreamController(*stream*, *controller*, *startAlgorithm*, *pullAlgorithm*, *cancelAlgorithm*, *highWaterMark*, *autoAllocateChunkSize*).
9. Return *stream*.

3.4.5. InitializeReadableStream (*stream*) nothrow §

1. Set *stream*.[[state]] to `"readable"`.
2. Set *stream*.[[reader]] and *stream*.[[storedError]] to **undefined**.
3. Set *stream*.[[disturbed]] to **false**.

3.4.6. IsReadableStream (*x*) nothrow §

1. If Type(*x*) is not Object, return **false**.
2. If *x* does not have a [[readableStreamController]] internal slot, return **false**.
3. Return **true**.

3.4.7. IsReadableStreamDisturbed (*stream*) nothrow §

This abstract operation is meant to be called from other specifications that may wish to query whether or not a readable stream has ever been read from or canceled.

1. Assert: ! IsReadableStream(*stream*) is **true**.
2. Return *stream*.[[disturbed]].

3.4.8. IsReadableStreamLocked (*stream*) nothrow §

This abstract operation is meant to be called from other specifications that may wish to query whether or not a readable stream is locked to a reader.

1. Assert: ! IsReadableStream(*stream*) is **true**.
2. If *stream*.[[reader]] is **undefined**, return **false**.
3. Return **true**.

3.4.9. IsReadableStreamAsyncIterator (*x*) nothrow §

1. If Type(*x*) is not Object, return **false**.
2. If *x* does not have a [[asyncIteratorReader]] internal slot, return **false**.
3. Return **true**.

3.4.10. ReadableStreamTee (*stream*, *cloneForBranch2*) throws §

This abstract operation is meant to be called from other specifications that may wish to tee a given readable stream.

The second argument, *cloneForBranch2*, governs whether or not the data from the original stream will be cloned (using HTML's serializable objects framework) before appearing in the second of the returned branches. This is useful for scenarios where both branches are to be consumed in such a way that they might otherwise interfere with each other, such as by transferring their chunks. However, it does introduce a noticeable asymmetry between the two branches, and limits the possible chunks to serializable ones. [HTML]

Note

In this standard ReadableStreamTee is always called with cloneForBranch2 *set to* ***false****; other specifications pass* ***true****.*

1. Assert: ! IsReadableStream(*stream*) is **true**.
2. Assert: Type(*cloneForBranch2*) is Boolean.
3. Let *reader* be ? AcquireReadableStreamDefaultReader(*stream*).
4. Let *reading* be **false**.
5. Let *canceled1* be **false**.
6. Let *canceled2* be **false**.
7. Let *reason1* be **undefined**.
8. Let *reason2* be **undefined**.
9. Let *branch1* be **undefined**.
10. Let *branch2* be **undefined**.
11. Let *cancelPromise* be a new promise.
12. Let *pullAlgorithm* be the following steps:
    a. If *reading* is **true**, return a promise resolved with **undefined**.
    b. Set *reading* to **true**.
    c. Let *readPromise* be the result of transforming ! ReadableStreamDefaultReaderRead(*reader*) with a fulfillment handler which takes the argument *result* and performs the following steps:
        i. Set *reading* to **false**.
        ii. Assert: Type(*result*) is Object.
        iii. Let *done* be ! Get(*result*, `"done"`).
        iv. Assert: Type(*done*) is Boolean.
        v. If *done* is **true**,
            1. If *canceled1* is **false**,
                a. Perform ! ReadableStreamDefaultControllerClose(*branch1*.[[readableStreamController]]).
            2. If *canceled2* is **false**,
                a. Perform ! ReadableStreamDefaultControllerClose(*branch2*.[[readableStreamController]]).
            3. Return.
        vi. Let *value* be ! Get(*result*, `"value"`).
        vii. Let *value1* and *value2* be *value*.
        viii. If *canceled2* is **false** and *cloneForBranch2* is **true**, set *value2* to ? StructuredDeserialize(? StructuredSerialize(*value2*), the current Realm Record).
        ix. If *canceled1* is **false**, perform ? ReadableStreamDefaultControllerEnqueue(*branch1*.[[readableStreamController]], *value1*).
        x. If *canceled2* is **false**, perform ? ReadableStreamDefaultControllerEnqueue(*branch2*.[[readableStreamController]], *value2*).
    d. Set *readPromise*.[[PromiseIsHandled]] to **true**.
    e. Return a promise resolved with **undefined**.
13. Let *cancel1Algorithm* be the following steps, taking a *reason* argument:
    a. Set *canceled1* to **true**.
    b. Set *reason1* to *reason*.
    c. If *canceled2* is **true**,
        i. Let *compositeReason* be ! CreateArrayFromList(« *reason1*, *reason2* »).
        ii. Let *cancelResult* be ! ReadableStreamCancel(*stream*, *compositeReason*).
        iii. Resolve *cancelPromise* with *cancelResult*.
    d. Return *cancelPromise*.
14. Let *cancel2Algorithm* be the following steps, taking a *reason* argument:
    a. Set *canceled2* to **true**.
    b. Set *reason2* to *reason*.
    c. If *canceled1* is **true**,
        i. Let *compositeReason* be ! CreateArrayFromList(« *reason1*, *reason2* »).
        ii. Let *cancelResult* be ! ReadableStreamCancel(*stream*, *compositeReason*).
        iii. Resolve *cancelPromise* with *cancelResult*.
    d. Return *cancelPromise*.
15. Let *startAlgorithm* be an algorithm that returns **undefined**.
16. Set *branch1* to ! CreateReadableStream(*startAlgorithm*, *pullAlgorithm*, *cancel1Algorithm*).
17. Set *branch2* to ! CreateReadableStream(*startAlgorithm*, *pullAlgorithm*, *cancel2Algorithm*).
18. Upon rejection of *reader*.[[closedPromise]] with reason *r*,
    a. Perform ! ReadableStreamDefaultControllerError(*branch1*.[[readableStreamController]], *r*).
    b. Perform ! ReadableStreamDefaultControllerError(*branch2*.[[readableStreamController]], *r*).
19. Return « *branch1*, *branch2* ».

3.4.11. ReadableStreamPipeTo (*source*, *dest*, *preventClose*, *preventAbort*, *preventCancel*, *signal*) nothrow §

1. Assert: ! IsReadableStream(*source*) is **true**.
2. Assert: ! IsWritableStream(*dest*) is **true**.
3. Assert: Type(*preventClose*) is Boolean, Type(*preventAbort*) is Boolean, and Type(*preventCancel*) is Boolean.
4. Assert: *signal* is **undefined** or *signal* is an instance of the `AbortSignal` interface.

File an issue about the selected text

5. Assert: ! IsReadableStreamLocked(*source*) is **false**.
6. Assert: ! IsWritableStreamLocked(*dest*) is **false**.
7. If ! IsReadableByteStreamController(*source*.[[readableStreamController]]) is **true**, let *reader* be either ! AcquireReadableStreamBYOBReader(*source*) or ! AcquireReadableStreamDefaultReader(*source*), at the user agent's discretion.
8. Otherwise, let *reader* be ! AcquireReadableStreamDefaultReader(*source*).
9. Let *writer* be ! AcquireWritableStreamDefaultWriter(*dest*).
10. Let *shuttingDown* be **false**.
11. Let *promise* be a new promise.
12. If *signal* is not **undefined**,
    a. Let *abortAlgorithm* be the following steps:
        i. Let *error* be a new "AbortError" DOMException.
        ii. Let *actions* be an empty ordered set.
        iii. If *preventAbort* is **false**, append the following action to *actions*:
            1. If *dest*.[[*state*]] is `"writable"`, return ! WritableStreamAbort(*dest*, *error*).
            2. Otherwise, return a promise resolved with **undefined**.
        iv. If *preventCancel* is **false**, append the following action action to *actions*:
            1. If *source*.[[*state*]] is `"readable"`, return ! ReadableStreamCancel(*source*, *error*).
            2. Otherwise, return a promise resolved with **undefined**.
        v. Shutdown with an action consisting of waiting for all of the actions in *actions*, and with *error*.
    b. If *signal*'s aborted flag is set, perform *abortAlgorithm* and return *promise*.
    c. Add *abortAlgorithm* to *signal*.
13. In parallel but not really; see #905, using *reader* and *writer*, read all chunks from *source* and write them to *dest*. Due to the locking provided by the reader and writer, the exact manner in which this happens is not observable to author code, and so there is flexibility in how this is done. The following constraints apply regardless of the exact algorithm used:
    - **Public API must not be used:** while reading or writing, or performing any of the operations below, the JavaScript-modifiable reader, writer, and stream APIs (i.e. methods on the appropriate prototypes) must not be used. Instead, the streams must be manipulated directly.
    - **Backpressure must be enforced:**
        - While WritableStreamDefaultWriterGetDesiredSize(*writer*) is ≤ **0** or is **null**, the user agent must not read from *reader*.
        - If *reader* is a BYOB reader, WritableStreamDefaultWriterGetDesiredSize(*writer*) should be used as a basis to determine the size of the chunks read from *reader*.

          Note

          It's frequently inefficient to read chunks that are too small or too large. Other information might be factored in to determine the optimal chunk size.

        - Reads or writes should not be delayed for reasons other than these backpressure signals.

          ¶ Example

          An implementation that waits for each write to successfully complete before proceeding to the next read/write operation violates this recommendation. In doing so, such an implementation makes the internal queue of *dest* useless, as it ensures *dest* always contains at most one queued chunk.

    - **Shutdown must stop activity:** if *shuttingDown* becomes **true**, the user agent must not initiate further reads from *reader*, and must only perform writes of already-read chunks, as described below. In particular, the user agent must check the below conditions before performing any reads or writes, since they might lead to immediate shutdown.
    - **Error and close states must be propagated:** the following conditions must be applied in order.
        a. **Errors must be propagated forward:** if *source*.[[state]] is or becomes `"errored"`, then
            i. If *preventAbort* is **false**, shutdown with an action of ! WritableStreamAbort(*dest*, *source*.[[storedError]]) and with *source*.[[storedError]].
            ii. Otherwise, shutdown with *source*.[[storedError]].
        b. **Errors must be propagated backward:** if *dest*.[[state]] is or becomes `"errored"`, then
            i. If *preventCancel* is **false**, shutdown with an action of ! ReadableStreamCancel(*source*, *dest*.[[storedError]]) and with *dest*.[[storedError]].
            ii. Otherwise, shutdown with *dest*.[[storedError]].
        c. **Closing must be propagated forward:** if *source*.[[state]] is or becomes `"closed"`, then
            i. If *preventClose* is **false**, shutdown with an action of ! WritableStreamDefaultWriterCloseWithErrorPropagation(*writer*).
            ii. Otherwise, shutdown.
        d. **Closing must be propagated backward:** if ! WritableStreamCloseQueuedOrInFlight(*dest*) is **true** or *dest*.[[state]] is `"closed"`, then
            i. Assert: no chunks have been read or written.
            ii. Let *destClosed* be a new **TypeError**.
            iii. If *preventCancel* is **false**, shutdown with an action of ! ReadableStreamCancel(*source*, *destClosed*) and with *destClosed*.
            iv. Otherwise, shutdown with *destClosed*.
    - *Shutdown with an action*: if any of the above requirements ask to shutdown with an action *action*, optionally with an error *originalError*, then:
        a. If *shuttingDown* is **true**, abort these substeps.
        b. Set *shuttingDown* to **true**.

File an issue about the selected text [state]] is `"writable"` and ! WritableStreamCloseQueuedOrInFlight(*dest*) is **false**,

i. If any chunks have been read but not yet written, write them to *dest*.
ii. Wait until every chunk that has been read has been written (i.e. the corresponding promises have settled).

d. Let *p* be the result of performing *action*.
e. Upon fulfillment of *p*, finalize, passing along *originalError* if it was given.
f. Upon rejection of *p* with reason *newError*, finalize with *newError*.

- *Shutdown*: if any of the above requirements or steps ask to shutdown, optionally with an error *error*, then:
  a. If *shuttingDown* is **true**, abort these substeps.
  b. Set *shuttingDown* to **true**.
  c. If *dest*.[[state]] is `"writable"` and ! WritableStreamCloseQueuedOrInFlight(*dest*) is **false**,
     i. If any chunks have been read but not yet written, write them to *dest*.
     ii. Wait until every chunk that has been read has been written (i.e. the corresponding promises have settled).
  d. Finalize, passing along *error* if it was given.
- *Finalize*: both forms of shutdown will eventually ask to finalize, optionally with an error *error*, which means to perform the following steps:
  a. Perform ! WritableStreamDefaultWriterRelease(*writer*).
  b. Perform ! ReadableStreamReaderGenericRelease(*reader*).
  c. If *signal* is not **undefined**, remove *abortAlgorithm* from *signal*.
  d. If *error* was given, reject *promise* with *error*.
  e. Otherwise, resolve *promise* with **undefined**.

14. Return *promise*.

Note

Various abstract operations performed here include object creation (often of promises), which usually would require specifying a realm for the created object. However, because of the locking, none of these objects can be observed by author code. As such, the realm used to create them does not matter.

3.5. The interface between readable streams and controllers §

In terms of specification factoring, the way that the `ReadableStream` class encapsulates the behavior of both simple readable streams and readable byte streams into a single class is by centralizing most of the potentially-varying logic inside the two controller classes, `ReadableStreamDefaultController` and `ReadableByteStreamController`. Those classes define most of the stateful internal slots and abstract operations for how a stream's internal queue is managed and how it interfaces with its underlying source or underlying byte source.

Each controller class defines two internal methods, which are called by the `ReadableStream` algorithms:

[[CancelSteps]](reason)
The controller's steps that run in reaction to the stream being canceled, used to clean up the state stored in the controller and inform the underlying source.

[[PullSteps]]()
The controller's steps that run when a default reader is read from, used to pull from the controller any queued chunks, or pull from the underlying source to get more chunks.

(These are defined as internal methods, instead of as abstract operations, so that they can be called polymorphically by the `ReadableStream` algorithms, without having to branch on which type of controller is present.)

The rest of this section concerns abstract operations that go in the other direction: they are used by the controller implementations to affect their associated `ReadableStream` object. This translates internal state changes of the controller into developer-facing results visible through the `ReadableStream`'s public API.

3.5.1. ReadableStreamAddReadIntoRequest (*stream*) nothrow §

1. Assert: ! IsReadableStreamBYOBReader(*stream*.[[reader]]) is **true**.
2. Assert: *stream*.[[state]] is `"readable"` or `"closed"`.
3. Let *promise* be a new promise.
4. Let *readIntoRequest* be Record {[[promise]]: *promise*}.
5. Append *readIntoRequest* as the last element of *stream*.[[reader]].[[readIntoRequests]].
6. Return *promise*.

3.5.2. ReadableStreamAddReadRequest (*stream*) nothrow §

1. Assert: ! IsReadableStreamDefaultReader(*stream*.[[reader]]) is **true**.

File an issue about the selected text ; `"readable"`.

3. Let *promise* be a new promise.
4. Let *readRequest* be Record {[[promise]]: *promise*}.
5. Append *readRequest* as the last element of *stream*.[[reader]].[[readRequests]].
6. Return *promise*.

3.5.3. ReadableStreamCancel (*stream*, *reason*) nothrow §

1. Set *stream*.[[disturbed]] to **true**.
2. If *stream*.[[state]] is `"closed"`, return a promise resolved with **undefined**.
3. If *stream*.[[state]] is `"errored"`, return a promise rejected with *stream*.[[storedError]].
4. Perform ! ReadableStreamClose(*stream*).
5. Let *sourceCancelPromise* be ! *stream*.[[readableStreamController]].[[CancelSteps]](*reason*).
6. Return the result of transforming *sourceCancelPromise* with a fulfillment handler that returns **undefined**.

3.5.4. ReadableStreamClose (*stream*) nothrow §

1. Assert: *stream*.[[state]] is `"readable"`.
2. Set *stream*.[[state]] to `"closed"`.
3. Let *reader* be *stream*.[[reader]].
4. If *reader* is **undefined**, return.
5. If ! IsReadableStreamDefaultReader(*reader*) is **true**,
    a. Repeat for each *readRequest* that is an element of *reader*.[[readRequests]],
        i. Resolve *readRequest*.[[promise]] with ! ReadableStreamCreateReadResult(**undefined**, **true**, *reader*.[[forAuthorCode]]).
    b. Set *reader*.[[readRequests]] to an empty List.
6. Resolve *reader*.[[closedPromise]] with **undefined**.

Note

The case where stream.*[[state]] is* `"closed"`*, but* stream.*[[closeRequested]] is* ***false****, will happen if the stream was closed without its controller's close method ever being called: i.e., if the stream was closed by a call to* `cancel(reason)`*. In this case we allow the controller's* `close` *method to be called and silently do nothing, since the cancelation was outside the control of the underlying source.*

3.5.5. ReadableStreamCreateReadResult (*value*, *done*, *forAuthorCode*) nothrow §

Note

When forAuthorCode *is* ***true****, this abstract operation gives the same result as CreateIterResultObject(*value, done*). This provides the expected semantics when the object is to be returned from the* `defaultReader.read()` *or* `byobReader.read()` *methods.*

However, resolving promises with such objects will unavoidably result in an access to `Object.prototype.then`*. For internal use, particularly in* `pipeTo()` *and in other specifications, it is important that reads not be observable by author code—even if that author code has tampered with* `Object.prototype`*. For this reason, a* ***false*** *value of* forAuthorCode *results in an object with a* ***null*** *prototype, keeping promise resolution unobservable.*

The underlying issue here is that reading from streams always uses promises for `{ value, done }` *objects, even in specifications. Although it is conceivable we could rephrase all of the internal algorithms to not use promises and not use JavaScript objects, and instead only package up the results into promise-for-*`{ value, done }` *when a* `read()` *method is called, this would be a large undertaking, which we have not done. See whatwg/infra#181 for more background on this subject.*

1. Let *prototype* be **null**.
2. If *forAuthorCode* is **true**, set *prototype* to %ObjectPrototype%.
3. Assert: Type(*done*) is Boolean.
4. Let *obj* be ObjectCreate(*prototype*).
5. Perform CreateDataProperty(*obj*, `"value"`, *value*).
6. Perform CreateDataProperty(*obj*, `"done"`, *done*).
7. Return *obj*.

3.5.6. ReadableStreamError (*stream*, *e*) nothrow §

1. Assert: ! IsReadableStream(*stream*) is **true**.
2. Assert: *stream*.[[state]] is `"readable"`.

File an issue about the selected text

3. Set *stream*.[[state]] to `"errored"`.
4. Set *stream*.[[storedError]] to *e*.
5. Let *reader* be *stream*.[[reader]].
6. If *reader* is **undefined**, return.
7. If ! IsReadableStreamDefaultReader(*reader*) is **true**,
    a. Repeat for each *readRequest* that is an element of *reader*.[[readRequests]],
        i. Reject *readRequest*.[[promise]] with *e*.
    b. Set *reader*.[[readRequests]] to a new empty List.
8. Otherwise,
    a. Assert: ! IsReadableStreamBYOBReader(*reader*).
    b. Repeat for each *readIntoRequest* that is an element of *reader*.[[readIntoRequests]],
        i. Reject *readIntoRequest*.[[promise]] with *e*.
    c. Set *reader*.[[readIntoRequests]] to a new empty List.
9. Reject *reader*.[[closedPromise]] with *e*.
10. Set *reader*.[[closedPromise]].[[PromiseIsHandled]] to **true**.

3.5.7. ReadableStreamFulfillReadIntoRequest (*stream*, *chunk*, *done*) nothrow §

1. Let *reader* be *stream*.[[reader]].
2. Let *readIntoRequest* be the first element of *reader*.[[readIntoRequests]].
3. Remove *readIntoRequest* from *reader*.[[readIntoRequests]], shifting all other elements downward (so that the second becomes the first, and so on).
4. Resolve *readIntoRequest*.[[promise]] with ! ReadableStreamCreateReadResult(*chunk*, *done*, *reader*.[[forAuthorCode]]).

3.5.8. ReadableStreamFulfillReadRequest (*stream*, *chunk*, *done*) nothrow §

1. Let *reader* be *stream*.[[reader]].
2. Let *readRequest* be the first element of *reader*.[[readRequests]].
3. Remove *readRequest* from *reader*.[[readRequests]], shifting all other elements downward (so that the second becomes the first, and so on).
4. Resolve *readRequest*.[[promise]] with ! ReadableStreamCreateReadResult(*chunk*, *done*, *reader*.[[forAuthorCode]]).

3.5.9. ReadableStreamGetNumReadIntoRequests (*stream*) nothrow §

1. Return the number of elements in *stream*.[[reader]].[[readIntoRequests]].

3.5.10. ReadableStreamGetNumReadRequests (*stream*) nothrow §

1. Return the number of elements in *stream*.[[reader]].[[readRequests]].

3.5.11. ReadableStreamHasBYOBReader (*stream*) nothrow §

1. Let *reader* be *stream*.[[reader]].
2. If *reader* is **undefined**, return **false**.
3. If ! IsReadableStreamBYOBReader(*reader*) is **false**, return **false**.
4. Return **true**.

3.5.12. ReadableStreamHasDefaultReader (*stream*) nothrow §

1. Let *reader* be *stream*.[[reader]].
2. If *reader* is **undefined**, return **false**.
3. If ! IsReadableStreamDefaultReader(*reader*) is **false**, return **false**.
4. Return **true**.

3.6. Class `ReadableStreamDefaultReader`

The `ReadableStreamDefaultReader` class represents a default reader designed to be vended by a `ReadableStream` instance.

3.6.1. Class definition §

This section is non-normative.

If one were to write the `ReadableStreamDefaultReader` class in something close to the syntax of [ECMASCRIPT], it would look like

```
class ReadableStreamDefaultReader {
  constructor(stream)

  get closed()

  cancel(reason)
  read()
  releaseLock()
}
```

3.6.2. Internal slots §

Instances of `ReadableStreamDefaultReader` are created with the internal slots described in the following table:

Internal Slot	Description (*non-normative*)
[[closedPromise]]	A promise returned by the reader's `closed` getter
[[forAuthorCode]]	A boolean flag indicating whether this reader is visible to author code
[[ownerReadableStream]]	A `ReadableStream` instance that owns this reader
[[readRequests]]	A List of promises returned by calls to the reader's `read()` method that have not yet been resolved, due to the consumer requesting chunks sooner than they are available; also used for the IsReadableStreamDefaultReader brand check

3.6.3. new ReadableStreamDefaultReader(*stream*)

Note

The `ReadableStreamDefaultReader` constructor is generally not meant to be used directly; instead, a stream's `getReader()` method ought to be used.

1. If ! IsReadableStream(*stream*) is **false**, throw a **TypeError** exception.
2. If ! IsReadableStreamLocked(*stream*) is **true**, throw a **TypeError** exception.
3. Perform ! ReadableStreamReaderGenericInitialize(**this**, *stream*).
4. Set **this**.[[readRequests]] to a new empty List.

3.6.4. Properties of the `ReadableStreamDefaultReader` prototype §

3.6.4.1. get closed

Note

The `closed` getter returns a promise that will be fulfilled when the stream becomes closed, or rejected if the stream ever errors or the reader's lock is released before the stream finishes closing.

1. If ! IsReadableStreamDefaultReader(**this**) is **false**, return a promise rejected with a **TypeError** exception.
2. Return **this**.[[closedPromise]].

3.6.4.2. cancel(*reason*)

Note

If the reader is active, the `cancel` method behaves the same as that for the associated stream.

File an issue about the selected text ;faultReader(**this**) is **false**, return a promise rejected with a **TypeError** exception.

2. If **this**.[[ownerReadableStream]] is **undefined**, return a promise rejected with a **TypeError** exception.
3. Return ! ReadableStreamReaderGenericCancel(**this**, *reason*).

3.6.4.3. read()

Note

The `read` method will return a promise that allows access to the next chunk from the stream's internal queue, if available.

- *If the chunk does become available, the promise will be fulfilled with an object of the form `{ value: theChunk, done: false }`.*
- *If the stream becomes closed, the promise will be fulfilled with an object of the form `{ value: undefined, done: true }`.*
- *If the stream becomes errored, the promise will be rejected with the relevant error.*

If reading a chunk causes the queue to become empty, more data will be pulled from the underlying source.

1. If ! IsReadableStreamDefaultReader(**this**) is **false**, return a promise rejected with a **TypeError** exception.
2. If **this**.[[ownerReadableStream]] is **undefined**, return a promise rejected with a **TypeError** exception.
3. Return ! ReadableStreamDefaultReaderRead(**this**).

3.6.4.4. releaseLock()

Note

The `releaseLock` method releases the reader's lock on the corresponding stream. After the lock is released, the reader is no longer active. If the associated stream is errored when the lock is released, the reader will appear errored in the same way from now on; otherwise, the reader will appear closed.

*A reader's lock cannot be released while it still has a pending read request, i.e., if a promise returned by the reader's `read()` method has not yet been settled. Attempting to do so will throw a **TypeError** and leave the reader locked to the stream.*

1. If ! IsReadableStreamDefaultReader(**this**) is **false**, throw a **TypeError** exception.
2. If **this**.[[ownerReadableStream]] is **undefined**, return.
3. If **this**.[[readRequests]] is not empty, throw a **TypeError** exception.
4. Perform ! ReadableStreamReaderGenericRelease(**this**).

3.7. Class `ReadableStreamBYOBReader`

The `ReadableStreamBYOBReader` class represents a BYOB reader designed to be vended by a `ReadableStream` instance.

3.7.1. Class definition §

This section is non-normative.

If one were to write the `ReadableStreamBYOBReader` class in something close to the syntax of [ECMASCRIPT], it would look like

```
class ReadableStreamBYOBReader {
  constructor(stream)

  get closed()

  cancel(reason)
  read(view)
  releaseLock()
}
```

3.7.2. Internal slots §

Instances of `ReadableStreamBYOBReader` are created with the internal slots described in the following table:

Internal Slot	Description (*non-normative*)
[[closedPromise]]	A promise returned by the reader's `closed` getter
[[forAuthorCode]]	A boolean flag indicating whether this reader is visible to author code
[[ownerReadableStream]]	A `ReadableStream` instance that owns this reader
[[readIntoRequests]]	A List of promises returned by calls to the reader's `read(view)` method that have not yet been resolved, due to the consumer requesting chunks sooner than they are available; also used for the IsReadableStreamBYOBReader brand check

3.7.3. new ReadableStreamBYOBReader(*stream*)

Note

The `ReadableStreamBYOBReader` constructor is generally not meant to be used directly; instead, a stream's `getReader()` method ought to be used.

1. If ! IsReadableStream(*stream*) is **false**, throw a **TypeError** exception.
2. If ! IsReadableByteStreamController(*stream*.[[readableStreamController]]) is **false**, throw a **TypeError** exception.
3. If ! IsReadableStreamLocked(*stream*) is **true**, throw a **TypeError** exception.
4. Perform ! ReadableStreamReaderGenericInitialize(**this**, *stream*).
5. Set **this**.[[readIntoRequests]] to a new empty List.

3.7.4. Properties of the `ReadableStreamBYOBReader` prototype §

3.7.4.1. get closed

Note

The `closed` getter returns a promise that will be fulfilled when the stream becomes closed, or rejected if the stream ever errors or the reader's lock is released before the stream finishes closing.

1. If ! IsReadableStreamBYOBReader(**this**) is **false**, return a promise rejected with a **TypeError** exception.
2. Return **this**.[[closedPromise]].

3.7.4.2. cancel(*reason*)

Note

If the reader is active, the `cancel` method behaves the same as that for the associated stream.

1. If ! IsReadableStreamBYOBReader(**this**) is **false**, return a promise rejected with a **TypeError** exception.
2. If **this**.[[ownerReadableStream]] is **undefined**, return a promise rejected with a **TypeError** exception.
3. Return ! ReadableStreamReaderGenericCancel(**this**, *reason*).

3.7.4.3. read(*view*)

Note

The `read` method will write read bytes into `view` and return a promise resolved with a possibly transferred buffer as described below.

- *If the chunk does become available, the promise will be fulfilled with an object of the form `{ value: theChunk, done: false }`.*
- *If the stream becomes closed, the promise will be fulfilled with an object of the form `{ value: undefined, done: true }`.*
- *If the stream becomes errored, the promise will be rejected with the relevant error.*

If reading a chunk causes the queue to become empty, more data will be pulled from the underlying byte source.

1. If ! IsReadableStreamBYOBReader(**this**) is **false**, return a promise rejected with a **TypeError** exception.
2. If **this**.[[ownerReadableStream]] is **undefined**, return a promise rejected with a **TypeError** exception.
3. If Type(*view*) is not Object, return a promise rejected with a **TypeError** exception.
4. If *view* does not have a [[ViewedArrayBuffer]] internal slot, return a promise rejected with a **TypeError** exception.
5. If ! IsDetachedBuffer(*view*.[[ViewedArrayBuffer]]) is **true**, return a promise rejected with a **TypeError** exception.
6. If *view*.[[ByteLength]] is **0**, return a promise rejected with a **TypeError** exception.
7. Return ! ReadableStreamBYOBReaderRead(**this**, *view*).

Note

The `releaseLock` method releases the reader's lock on the corresponding stream. After the lock is released, the reader is no longer active. If the associated stream is errored when the lock is released, the reader will appear errored in the same way from now on; otherwise, the reader will appear closed.

*A reader's lock cannot be released while it still has a pending read request, i.e., if a promise returned by the reader's `read()` method has not yet been settled. Attempting to do so will throw a **TypeError** and leave the reader locked to the stream.*

1. If ! IsReadableStreamBYOBReader(**this**) is **false**, throw a **TypeError** exception.
2. If **this**.[[ownerReadableStream]] is **undefined**, return.
3. If **this**.[[readIntoRequests]] is not empty, throw a **TypeError** exception.
4. Perform ! ReadableStreamReaderGenericRelease(**this**).

3.8. Readable stream reader abstract operations §

3.8.1. IsReadableStreamDefaultReader (*x*) nothrow §

1. If Type(*x*) is not Object, return **false**.
2. If *x* does not have a [[readRequests]] internal slot, return **false**.
3. Return **true**.

3.8.2. IsReadableStreamBYOBReader (*x*) nothrow §

1. If Type(*x*) is not Object, return **false**.
2. If *x* does not have a [[readIntoRequests]] internal slot, return **false**.
3. Return **true**.

3.8.3. ReadableStreamReaderGenericCancel (*reader*, *reason*) nothrow §

1. Let *stream* be *reader*.[[ownerReadableStream]].
2. Assert: *stream* is not **undefined**.
3. Return ! ReadableStreamCancel(*stream*, *reason*).

3.8.4. ReadableStreamReaderGenericInitialize (*reader*, *stream*) nothrow §

1. Set *reader*.[[forAuthorCode]] to **true**.
2. Set *reader*.[[ownerReadableStream]] to *stream*.
3. Set *stream*.[[reader]] to *reader*.
4. If *stream*.[[state]] is `"readable"`,
    a. Set *reader*.[[closedPromise]] to a new promise.
5. Otherwise, if *stream*.[[state]] is `"closed"`,
    a. Set *reader*.[[closedPromise]] to a promise resolved with **undefined**.
6. Otherwise,
    a. Assert: *stream*.[[state]] is `"errored"`.
    b. Set *reader*.[[closedPromise]] to a promise rejected with *stream*.[[storedError]].
    c. Set *reader*.[[closedPromise]].[[PromiseIsHandled]] to **true**.

3.8.5. ReadableStreamReaderGenericRelease (*reader*) nothrow §

1. Assert: *reader*.[[ownerReadableStream]] is not **undefined**.
2. Assert: *reader*.[[ownerReadableStream]].[[reader]] is *reader*.
3. If *reader*.[[ownerReadableStream]].[[state]] is `"readable"`, reject *reader*.[[closedPromise]] with a **TypeError** exception.
4. Otherwise, set *reader*.[[closedPromise]] to a promise rejected with a **TypeError** exception.
5. Set *reader*.[[closedPromise]].[[PromiseIsHandled]] to **true**.
6. Set *reader*.[[ownerReadableStream]].[[reader]] to **undefined**.
7. Set *reader*.[[ownerReadableStream]] to **undefined**.

3.8.6. ReadableStreamBYOBReaderRead (*reader*, *view*) nothrow §

1. Let *stream* be *reader*.[[ownerReadableStream]].
2. Assert: *stream* is not **undefined**.
3. Set *stream*.[[disturbed]] to **true**.
4. If *stream*.[[state]] is `"errored"`, return a promise rejected with *stream*.[[storedError]].
5. Return ! ReadableByteStreamControllerPullInto(*stream*.[[readableStreamController]], *view*).

3.8.7. ReadableStreamDefaultReaderRead (*reader*) nothrow §

1. Let *stream* be *reader*.[[ownerReadableStream]].
2. Assert: *stream* is not **undefined**.
3. Set *stream*.[[disturbed]] to **true**.
4. If *stream*.[[state]] is `"closed"`, return a promise resolved with ! ReadableStreamCreateReadResult(**undefined**, **true**, *reader*.[[forAuthorCode]]).
5. If *stream*.[[state]] is `"errored"`, return a promise rejected with *stream*.[[storedError]].
6. Assert: *stream*.[[state]] is `"readable"`.
7. Return ! *stream*.[[readableStreamController]].[[PullSteps]]().

3.9. Class `ReadableStreamDefaultController`

The `ReadableStreamDefaultController` class has methods that allow control of a `ReadableStream`'s state and internal queue. When constructing a `ReadableStream` that is not a readable byte stream, the underlying source is given a corresponding `ReadableStreamDefaultController` instance to manipulate.

3.9.1. Class definition §

This section is non-normative.

If one were to write the `ReadableStreamDefaultController` class in something close to the syntax of [ECMASCRIPT], it would look like

```
class ReadableStreamDefaultController {
  constructor() // always throws

  get desiredSize()

  close()
  enqueue(chunk)
  error(e)
}
```

3.9.2. Internal slots §

Instances of `ReadableStreamDefaultController` are created with the internal slots described in the following table:

Internal Slot	Description (*non-normative*)
[[cancelAlgorithm]]	A promise-returning algorithm, taking one argument (the cancel reason), which communicates a requested cancelation to the underlying source
[[closeRequested]]	A boolean flag indicating whether the stream has been closed by its underlying source, but still has chunks in its internal queue that have not yet been read
[[controlledReadableStream]]	The `ReadableStream` instance controlled
[[pullAgain]]	A boolean flag set to **true** if the stream's mechanisms requested a call to the underlying source's pull algorithm to pull more data, but the pull could not yet be done since a previous call is still executing
[[pullAlgorithm]]	A promise-returning algorithm that pulls data from the underlying source
[[pulling]]	A boolean flag set to **true** while the underlying source's pull algorithm is executing and the returned promise has not yet fulfilled, used to prevent reentrant calls
[[queue]]	A List representing the stream's internal queue of chunks
[[queueTotalSize]]	The total size of all the chunks stored in [[queue]] (see §6.2 Queue-with-sizes operations)
[[started]]	A boolean flag indicating whether the underlying source has finished starting
[[strategyHWM]]	A number supplied to the constructor as part of the stream's queuing strategy, indicating the point at which the stream will apply backpressure to its underlying source
[[strategySizeAlgorithm]]	An algorithm to calculate the size of enqueued chunks, as part of the stream's queuing strategy

3.9.3. new ReadableStreamDefaultController()

Note

The `ReadableStreamDefaultController` constructor cannot be used directly; `ReadableStreamDefaultController` instances are created automatically during `ReadableStream` construction.

1. Throw a **TypeError**.

3.9.4. Properties of the `ReadableStreamDefaultController` prototype §

3.9.4.1. get desiredSize

Note

The `desiredSize` getter returns the desired size to fill the controlled stream's internal queue. It can be negative, if the queue is over-full. An underlying source ought to use this information to determine when and how to apply backpressure.

1. If ! IsReadableStreamDefaultController(**this**) is **false**, throw a **TypeError** exception.
2. Return ! ReadableStreamDefaultControllerGetDesiredSize(**this**).

3.9.4.2. close()

Note

The `close` method will close the controlled readable stream. Consumers will still be able to read any previously-enqueued chunks from the stream, but once those are read, the stream will become closed.

1. If ! IsReadableStreamDefaultController(**this**) is **false**, throw a **TypeError** exception.
2. If ! ReadableStreamDefaultControllerCanCloseOrEnqueue(**this**) is **false**, throw a **TypeError** exception.
3. Perform ! ReadableStreamDefaultControllerClose(**this**).

3.9.4.3. enqueue(*chunk*)

Note

The `enqueue` method will enqueue a given chunk in the controlled readable stream.

1. If ! IsReadableStreamDefaultController(**this**) is **false**, throw a **TypeError** exception.
2. If ! ReadableStreamDefaultControllerCanCloseOrEnqueue(**this**) is **false**, throw a **TypeError** exception.
3. Return ? ReadableStreamDefaultControllerEnqueue(**this**, *chunk*).

3.9.4.4. error(*e*)

Note

The `error` method will error the readable stream, making all future interactions with it fail with the given error `e`.

1. If ! IsReadableStreamDefaultController(**this**) is **false**, throw a **TypeError** exception.
2. Perform ! ReadableStreamDefaultControllerError(**this**, *e*).

3.9.5. Readable stream default controller internal methods §

The following are additional internal methods implemented by each `ReadableStreamDefaultController` instance. The readable stream implementation will polymorphically call to either these or their counterparts for BYOB controllers.

3.9.5.1. [[CancelSteps]](*reason*) §

1. Perform ! ResetQueue(**this**).
2. Let *result* be the result of performing **this**.[[cancelAlgorithm]], passing *reason*.
3. Perform ! ReadableStreamDefaultControllerClearAlgorithms(**this**).
4. Return *result*.

File an issue about the selected text

3.9.5.2. [[PullSteps]]( ) §

1. Let *stream* be **this**.[[controlledReadableStream]].
2. If **this**.[[queue]] is not empty,
    a. Let *chunk* be ! DequeueValue(**this**).
    b. If **this**.[[closeRequested]] is **true** and **this**.[[queue]] is empty,
        i. Perform ! ReadableStreamDefaultControllerClearAlgorithms(**this**).
        ii. Perform ! ReadableStreamClose(*stream*).
    c. Otherwise, perform ! ReadableStreamDefaultControllerCallPullIfNeeded(**this**).
    d. Return a promise resolved with ! ReadableStreamCreateReadResult(*chunk*, **false**, *stream*.[[reader]].[[forAuthorCode]]).
3. Let *pendingPromise* be ! ReadableStreamAddReadRequest(*stream*).
4. Perform ! ReadableStreamDefaultControllerCallPullIfNeeded(**this**).
5. Return *pendingPromise*.

3.10. Readable stream default controller abstract operations §

3.10.1. IsReadableStreamDefaultController (*x*) nothrow §

1. If Type(*x*) is not Object, return **false**.
2. If *x* does not have a [[controlledReadableStream]] internal slot, return **false**.
3. Return **true**.

3.10.2. ReadableStreamDefaultControllerCallPullIfNeeded (*controller*) nothrow §

1. Let *shouldPull* be ! ReadableStreamDefaultControllerShouldCallPull(*controller*).
2. If *shouldPull* is **false**, return.
3. If *controller*.[[pulling]] is **true**,
    a. Set *controller*.[[pullAgain]] to **true**.
    b. Return.
4. Assert: *controller*.[[pullAgain]] is **false**.
5. Set *controller*.[[pulling]] to **true**.
6. Let *pullPromise* be the result of performing *controller*.[[pullAlgorithm]].
7. Upon fulfillment of *pullPromise*,
    a. Set *controller*.[[pulling]] to **false**.
    b. If *controller*.[[pullAgain]] is **true**,
        i. Set *controller*.[[pullAgain]] to **false**.
        ii. Perform ! ReadableStreamDefaultControllerCallPullIfNeeded(*controller*).
8. Upon rejection of *pullPromise* with reason *e*,
    a. Perform ! ReadableStreamDefaultControllerError(*controller*, *e*).

3.10.3. ReadableStreamDefaultControllerShouldCallPull (*controller*) nothrow §

1. Let *stream* be *controller*.[[controlledReadableStream]].
2. If ! ReadableStreamDefaultControllerCanCloseOrEnqueue(*controller*) is **false**, return **false**.
3. If *controller*.[[started]] is **false**, return **false**.
4. If ! IsReadableStreamLocked(*stream*) is **true** and ! ReadableStreamGetNumReadRequests(*stream*) > **0**, return **true**.
5. Let *desiredSize* be ! ReadableStreamDefaultControllerGetDesiredSize(*controller*).
6. Assert: *desiredSize* is not **null**.
7. If *desiredSize* > **0**, return **true**.
8. Return **false**.

3.10.4. ReadableStreamDefaultControllerClearAlgorithms (*controller*) nothrow §

This abstract operation is called once the stream is closed or errored and the algorithms will not be executed any more. By removing the algorithm references it permits the underlying source object to be garbage collected even if the `ReadableStream` itself is still referenced.

Note

The results of this algorithm are not currently observable, but could become so if JavaScript eventually adds weak references. But even without that factor, implementations will likely want to include similar steps.

File an issue about the selected text thm]] to **undefined**.

2. Set *controller*.[[cancelAlgorithm]] to **undefined**.
3. Set *controller*.[[strategySizeAlgorithm]] to **undefined**.

3.10.5. ReadableStreamDefaultControllerClose (*controller*) nothrow §

This abstract operation can be called by other specifications that wish to close a readable stream, in the same way a developer-created stream would be closed by its associated controller object. Specifications should *not* do this to streams they did not create, and must ensure they have obeyed the preconditions (listed here as asserts).

1. Let *stream* be *controller*.[[controlledReadableStream]].
2. Assert: ! ReadableStreamDefaultControllerCanCloseOrEnqueue(*controller*) is **true**.
3. Set *controller*.[[closeRequested]] to **true**.
4. If *controller*.[[queue]] is empty,
    a. Perform ! ReadableStreamDefaultControllerClearAlgorithms(*controller*).
    b. Perform ! ReadableStreamClose(*stream*).

3.10.6. ReadableStreamDefaultControllerEnqueue (*controller*, *chunk*) throws §

This abstract operation can be called by other specifications that wish to enqueue chunks in a readable stream, in the same way a developer would enqueue chunks using the stream's associated controller object. Specifications should *not* do this to streams they did not create, and must ensure they have obeyed the preconditions (listed here as asserts).

1. Let *stream* be *controller*.[[controlledReadableStream]].
2. Assert: ! ReadableStreamDefaultControllerCanCloseOrEnqueue(*controller*) is **true**.
3. If ! IsReadableStreamLocked(*stream*) is **true** and ! ReadableStreamGetNumReadRequests(*stream*) > **0**, perform ! ReadableStreamFulfillReadRequest(*stream*, *chunk*, **false**).
4. Otherwise,
    a. Let *result* be the result of performing *controller*.[[strategySizeAlgorithm]], passing in *chunk*, and interpreting the result as an ECMAScript completion value.
    b. If *result* is an abrupt completion,
        i. Perform ! ReadableStreamDefaultControllerError(*controller*, *result*.[[Value]]).
        ii. Return *result*.
    c. Let *chunkSize* be *result*.[[Value]].
    d. Let *enqueueResult* be EnqueueValueWithSize(*controller*, *chunk*, *chunkSize*).
    e. If *enqueueResult* is an abrupt completion,
        i. Perform ! ReadableStreamDefaultControllerError(*controller*, *enqueueResult*.[[Value]]).
        ii. Return *enqueueResult*.
5. Perform ! ReadableStreamDefaultControllerCallPullIfNeeded(*controller*).

3.10.7. ReadableStreamDefaultControllerError (*controller*, *e*) nothrow §

This abstract operation can be called by other specifications that wish to move a readable stream to an errored state, in the same way a developer would error a stream using its associated controller object. Specifications should *not* do this to streams they did not create.

1. Let *stream* be *controller*.[[controlledReadableStream]].
2. If *stream*.[[state]] is not `"readable"`, return.
3. Perform ! ResetQueue(*controller*).
4. Perform ! ReadableStreamDefaultControllerClearAlgorithms(*controller*).
5. Perform ! ReadableStreamError(*stream*, *e*).

3.10.8. ReadableStreamDefaultControllerGetDesiredSize (*controller*) nothrow §

This abstract operation can be called by other specifications that wish to determine the desired size to fill this stream's internal queue, similar to how a developer would consult the `desiredSize` property of the stream's associated controller object. Specifications should *not* use this on streams they did not create.

1. Let *stream* be *controller*.[[controlledReadableStream]].
2. Let *state* be *stream*.[[state]].
3. If *state* is `"errored"`, return **null**.
4. If *state* is `"closed"`, return **0**.

File an issue about the selected text gyHWM]] − *controller*.[[queueTotalSize]].

3.10.9. ReadableStreamDefaultControllerHasBackpressure (*controller*) nothrow §

This abstract operation is used in the implementation of TransformStream.

1. If ! ReadableStreamDefaultControllerShouldCallPull(*controller*) is **true**, return **false**.
2. Otherwise, return **true**.

3.10.10. ReadableStreamDefaultControllerCanCloseOrEnqueue (*controller*) nothrow §

1. Let *state* be *controller*.[[controlledReadableStream]].[[state]].
2. If *controller*.[[closeRequested]] is **false** and *state* is `"readable"`, return **true**.
3. Otherwise, return **false**.

Note

The case where stream.*[[closeRequested]] is* ***false****, but* stream.*[[state]] is not* `"readable"`*, happens when the stream is errored via* `error(e)`*, or when it is closed without its controller's* `close` *method ever being called: e.g., if the stream was closed by a call to* `cancel(reason)`*.*

3.10.11. SetUpReadableStreamDefaultController(*stream*, *controller*, *startAlgorithm*, *pullAlgorithm*, *cancelAlgorithm*, *highWaterMark*, *sizeAlgorithm*) throws §

1. Assert: *stream*.[[readableStreamController]] is **undefined**.
2. Set *controller*.[[controlledReadableStream]] to *stream*.
3. Set *controller*.[[queue]] and *controller*.[[queueTotalSize]] to **undefined**, then perform ! ResetQueue(*controller*).
4. Set *controller*.[[started]], *controller*.[[closeRequested]], *controller*.[[pullAgain]], and *controller*.[[pulling]] to **false**.
5. Set *controller*.[[strategySizeAlgorithm]] to *sizeAlgorithm* and *controller*.[[strategyHWM]] to *highWaterMark*.
6. Set *controller*.[[pullAlgorithm]] to *pullAlgorithm*.
7. Set *controller*.[[cancelAlgorithm]] to *cancelAlgorithm*.
8. Set *stream*.[[readableStreamController]] to *controller*.
9. Let *startResult* be the result of performing *startAlgorithm*. (This may throw an exception.)
10. Let *startPromise* be a promise resolved with *startResult*.
11. Upon fulfillment of *startPromise*,
    a. Set *controller*.[[started]] to **true**.
    b. Assert: *controller*.[[pulling]] is **false**.
    c. Assert: *controller*.[[pullAgain]] is **false**.
    d. Perform ! ReadableStreamDefaultControllerCallPullIfNeeded(*controller*).
12. Upon rejection of *startPromise* with reason *r*,
    a. Perform ! ReadableStreamDefaultControllerError(*controller*, *r*).

3.10.12. SetUpReadableStreamDefaultControllerFromUnderlyingSource(*stream*, *underlyingSource*, *highWaterMark*, *sizeAlgorithm*) throws §

1. Assert: *underlyingSource* is not **undefined**.
2. Let *controller* be ObjectCreate(the original value of `ReadableStreamDefaultController`'s `prototype` property).
3. Let *startAlgorithm* be the following steps:
    a. Return ? InvokeOrNoop(*underlyingSource*, `"start"`, « *controller* »).
4. Let *pullAlgorithm* be ? CreateAlgorithmFromUnderlyingMethod(*underlyingSource*, `"pull"`, **0**, « *controller* »).
5. Let *cancelAlgorithm* be ? CreateAlgorithmFromUnderlyingMethod(*underlyingSource*, `"cancel"`, **1**, « »).
6. Perform ? SetUpReadableStreamDefaultController(*stream*, *controller*, *startAlgorithm*, *pullAlgorithm*, *cancelAlgorithm*, *highWaterMark*, *sizeAlgorithm*).

3.11. Class `ReadableByteStreamController`

The `ReadableByteStreamController` class has methods that allow control of a `ReadableStream`'s state and internal queue. When constructing a `ReadableStream`, the underlying byte source is given a corresponding `ReadableByteStreamController` instance to manipulate.

3.11.1. Class definition §

File an issue about the selected text

If one were to write the ReadableByteStreamController class in something close to the syntax of [ECMASCRIPT], it would look like

```
class ReadableByteStreamController {
  constructor() // always throws

  get byobRequest()
  get desiredSize()

  close()
  enqueue(chunk)
  error(e)
}
```

3.11.2. Internal slots §

Instances of ReadableByteStreamController are created with the internal slots described in the following table:

Internal Slot	Description (*non-normative*)
[[autoAllocateChunkSize]]	A positive integer, when the automatic buffer allocation feature is enabled. In that case, this value specifies the size of buffer to allocate. It is **undefined** otherwise.
[[byobRequest]]	A ReadableStreamBYOBRequest instance representing the current BYOB pull request, or **undefined** if there are no pending requests
[[cancelAlgorithm]]	A promise-returning algorithm, taking one argument (the cancel reason), which communicates a requested cancelation to the underlying source
[[closeRequested]]	A boolean flag indicating whether the stream has been closed by its underlying byte source, but still has chunks in its internal queue that have not yet been read
[[controlledReadableByteStream]]	The ReadableStream instance controlled
[[pullAgain]]	A boolean flag set to **true** if the stream's mechanisms requested a call to the underlying byte source's pull() method to pull more data, but the pull could not yet be done since a previous call is still executing
[[pullAlgorithm]]	A promise-returning algorithm that pulls data from the underlying source
[[pulling]]	A boolean flag set to **true** while the underlying byte source's pull() method is executing and has not yet fulfilled, used to prevent reentrant calls
[[pendingPullIntos]]	A List of descriptors representing pending BYOB pull requests
[[queue]]	A List representing the stream's internal queue of chunks
[[queueTotalSize]]	The total size (in bytes) of all the chunks stored in [[queue]]
[[started]]	A boolean flag indicating whether the underlying source has finished starting
[[strategyHWM]]	A number supplied to the constructor as part of the stream's queuing strategy, indicating the point at which the stream will apply backpressure to its underlying byte source

Note

Although ReadableByteStreamController instances have [[queue]] and [[queueTotalSize]] slots, we do not use most of the abstract operations in §6.2 Queue-with-sizes operations on them, as the way in which we manipulate this queue is rather different than the others in the spec. Instead, we update the two slots together manually.

This might be cleaned up in a future spec refactoring.

3.11.3. new ReadableByteStreamController()

Note

The ReadableByteStreamController constructor cannot be used directly; ReadableByteStreamController instances are created automatically during ReadableStream construction.

1. Throw a **TypeError** exception.

3.11.4. Properties of the ReadableByteStreamController prototype §

3.11.4.1. get byobRequest

Note

The byobRequest getter returns the current BYOB pull request.

1. If ! IsReadableByteStreamController(**this**) is **false**, throw a **TypeError** exception.
2. If **this**.[[byobRequest]] is **undefined** and **this**.[[pendingPullIntos]] is not empty,

File an issue about the selected text

a. Let *firstDescriptor* be the first element of **this**.[[pendingPullIntos]].
b. Let *view* be ! Construct(%Uint8Array%, « *firstDescriptor*.[[buffer]], *firstDescriptor*.[[byteOffset]] + *firstDescriptor*.[[bytesFilled]], *firstDescriptor*.[[byteLength]] − *firstDescriptor*.[[bytesFilled]] »).
c. Let *byobRequest* be ObjectCreate(the original value of `ReadableStreamBYOBRequest`'s `prototype` property).
d. Perform ! SetUpReadableStreamBYOBRequest(*byobRequest*, **this**, *view*).
e. Set **this**.[[byobRequest]] to *byobRequest*.

3. Return **this**.[[byobRequest]].

3.11.4.2. get desiredSize

Note

The `desiredSize` getter returns the desired size to fill the controlled stream's internal queue. It can be negative, if the queue is over-full. An underlying source ought to use this information to determine when and how to apply backpressure.

1. If ! IsReadableByteStreamController(**this**) is **false**, throw a **TypeError** exception.
2. Return ! ReadableByteStreamControllerGetDesiredSize(**this**).

3.11.4.3. close()

Note

The `close` method will close the controlled readable stream. Consumers will still be able to read any previously-enqueued chunks from the stream, but once those are read, the stream will become closed.

1. If ! IsReadableByteStreamController(**this**) is **false**, throw a **TypeError** exception.
2. If **this**.[[closeRequested]] is **true**, throw a **TypeError** exception.
3. If **this**.[[controlledReadableByteStream]].[[state]] is not `"readable"`, throw a **TypeError** exception.
4. Perform ? ReadableByteStreamControllerClose(**this**).

3.11.4.4. enqueue(*chunk*)

Note

The `enqueue` method will enqueue a given chunk in the controlled readable stream.

1. If ! IsReadableByteStreamController(**this**) is **false**, throw a **TypeError** exception.
2. If **this**.[[closeRequested]] is **true**, throw a **TypeError** exception.
3. If **this**.[[controlledReadableByteStream]].[[state]] is not `"readable"`, throw a **TypeError** exception.
4. If Type(*chunk*) is not Object, throw a **TypeError** exception.
5. If *chunk* does not have a [[ViewedArrayBuffer]] internal slot, throw a **TypeError** exception.
6. If ! IsDetachedBuffer(*chunk*.[[ViewedArrayBuffer]]) is **true**, throw a **TypeError** exception.
7. Return ! ReadableByteStreamControllerEnqueue(**this**, *chunk*).

3.11.4.5. error(*e*)

Note

The `error` method will error the readable stream, making all future interactions with it fail with the given error e.

1. If ! IsReadableByteStreamController(**this**) is **false**, throw a **TypeError** exception.
2. Perform ! ReadableByteStreamControllerError(**this**, *e*).

3.11.5. Readable stream BYOB controller internal methods §

The following are additional internal methods implemented by each `ReadableByteStreamController` instance. The readable stream implementation will polymorphically call to either these or their counterparts for default controllers.

3.11.5.1. [[CancelSteps]](*reason*) §

1. If **this**.[[pendingPullIntos]] is not empty,
   a. Let *firstDescriptor* be the first element of **this**.[[pendingPullIntos]].

    b. Set *firstDescriptor*.[[bytesFilled]] to **0**.
2. Perform ! ResetQueue(**this**).
3. Let *result* be the result of performing **this**.[[cancelAlgorithm]], passing in *reason*.
4. Perform ! ReadableByteStreamControllerClearAlgorithms(**this**).
5. Return *result*.

3.11.5.2. [[PullSteps]]( ) §

1. Let *stream* be **this**.[[controlledReadableByteStream]].
2. Assert: ! ReadableStreamHasDefaultReader(*stream*) is **true**.
3. If **this**.[[queueTotalSize]] > **0**,
    a. Assert: ! ReadableStreamGetNumReadRequests(*stream*) is **0**.
    b. Let *entry* be the first element of **this**.[[queue]].
    c. Remove *entry* from **this**.[[queue]], shifting all other elements downward (so that the second becomes the first, and so on).
    d. Set **this**.[[queueTotalSize]] to **this**.[[queueTotalSize]] − *entry*.[[byteLength]].
    e. Perform ! ReadableByteStreamControllerHandleQueueDrain(**this**).
    f. Let *view* be ! Construct(%Uint8Array%, « *entry*.[[buffer]], *entry*.[[byteOffset]], *entry*.[[byteLength]] »).
    g. Return a promise resolved with ! ReadableStreamCreateReadResult(*view*, **false**, *stream*.[[reader]].[[forAuthorCode]]).
4. Let *autoAllocateChunkSize* be **this**.[[autoAllocateChunkSize]].
5. If *autoAllocateChunkSize* is not **undefined**,
    a. Let *buffer* be Construct(%ArrayBuffer%, « *autoAllocateChunkSize* »).
    b. If *buffer* is an abrupt completion, return a promise rejected with *buffer*.[[Value]].
    c. Let *pullIntoDescriptor* be Record {[[buffer]]: *buffer*.[[Value]], [[byteOffset]]: **0**, [[byteLength]]: *autoAllocateChunkSize*, [[bytesFilled]]: **0**, [[elementSize]]: **1**, [[ctor]]: %Uint8Array%, [[readerType]]: `"default"`}.
    d. Append *pullIntoDescriptor* as the last element of **this**.[[pendingPullIntos]].
6. Let *promise* be ! ReadableStreamAddReadRequest(*stream*).
7. Perform ! ReadableByteStreamControllerCallPullIfNeeded(**this**).
8. Return *promise*.

3.12. Class `ReadableStreamBYOBRequest`

The `ReadableStreamBYOBRequest` class represents a pull into request in a `ReadableByteStreamController`.

3.12.1. Class definition §

This section is non-normative.

If one were to write the `ReadableStreamBYOBRequest` class in something close to the syntax of [ECMASCRIPT], it would look like

```
class ReadableStreamBYOBRequest {
  constructor(controller, view)

  get view()

  respond(bytesWritten)
  respondWithNewView(view)
}
```

3.12.2. Internal slots §

Instances of `ReadableStreamBYOBRequest` are created with the internal slots described in the following table:

Internal Slot	Description (*non-normative*)
[[associatedReadableByteStreamController]]	The parent `ReadableByteStreamController` instance
[[view]]	A typed array representing the destination region to which the controller can write generated data

3.12.3. new ReadableStreamBYOBRequest()

File an issue about the selected text

1. Throw a **TypeError** exception.

3.12.4. Properties of the `ReadableStreamBYOBRequest` prototype §

3.12.4.1. get view

1. If ! IsReadableStreamBYOBRequest(**this**) is **false**, throw a **TypeError** exception.
2. Return **this**.[[view]].

3.12.4.2. respond(*bytesWritten*)

1. If ! IsReadableStreamBYOBRequest(**this**) is **false**, throw a **TypeError** exception.
2. If **this**.[[associatedReadableByteStreamController]] is **undefined**, throw a **TypeError** exception.
3. If ! IsDetachedBuffer(**this**.[[view]].[[ViewedArrayBuffer]]) is **true**, throw a **TypeError** exception.
4. Return ? ReadableByteStreamControllerRespond(**this**.[[associatedReadableByteStreamController]], *bytesWritten*).

3.12.4.3. respondWithNewView(*view*)

1. If ! IsReadableStreamBYOBRequest(**this**) is **false**, throw a **TypeError** exception.
2. If **this**.[[associatedReadableByteStreamController]] is **undefined**, throw a **TypeError** exception.
3. If Type(*view*) is not Object, throw a **TypeError** exception.
4. If *view* does not have a [[ViewedArrayBuffer]] internal slot, throw a **TypeError** exception.
5. If ! IsDetachedBuffer(*view*.[[ViewedArrayBuffer]]) is **true**, throw a **TypeError** exception.
6. Return ? ReadableByteStreamControllerRespondWithNewView(**this**.[[associatedReadableByteStreamController]], *view*).

3.13. Readable stream BYOB controller abstract operations §

3.13.1. IsReadableStreamBYOBRequest (*x*) nothrow §

1. If Type(*x*) is not Object, return **false**.
2. If *x* does not have an [[associatedReadableByteStreamController]] internal slot, return **false**.
3. Return **true**.

3.13.2. IsReadableByteStreamController (*x*) nothrow §

1. If Type(*x*) is not Object, return **false**.
2. If *x* does not have an [[controlledReadableByteStream]] internal slot, return **false**.
3. Return **true**.

3.13.3. ReadableByteStreamControllerCallPullIfNeeded (*controller*) nothrow §

1. Let *shouldPull* be ! ReadableByteStreamControllerShouldCallPull(*controller*).
2. If *shouldPull* is **false**, return.
3. If *controller*.[[pulling]] is **true**,
    a. Set *controller*.[[pullAgain]] to **true**.
    b. Return.
4. Assert: *controller*.[[pullAgain]] is **false**.
5. Set *controller*.[[pulling]] to **true**.
6. Let *pullPromise* be the result of performing *controller*.[[pullAlgorithm]].
7. Upon fulfillment of *pullPromise*,
    a. Set *controller*.[[pulling]] to **false**.
    b. If *controller*.[[pullAgain]] is **true**,
        i. Set *controller*.[[pullAgain]] to **false**.
        ii. Perform ! ReadableByteStreamControllerCallPullIfNeeded(*controller*).
8. Upon rejection of *pullPromise* with reason *e*,
    a. Perform ! ReadableByteStreamControllerError(*controller*, *e*).

3.13.4. ReadableByteStreamControllerClearAlgorithms (*controller*) throws §

This abstract operation is called once the stream is closed or errored and the algorithms will not be executed any more. By removing the algorithm references it permits the underlying source object to be garbage collected even if the `ReadableStream` itself is still referenced.

Note

The results of this algorithm are not currently observable, but could become so if JavaScript eventually adds weak references. But even without that factor, implementations will likely want to include similar steps.

1. Set *controller*.[[pullAlgorithm]] to **undefined**.
2. Set *controller*.[[cancelAlgorithm]] to **undefined**.

3.13.5. ReadableByteStreamControllerClearPendingPullIntos (*controller*) nothrow §

1. Perform ! ReadableByteStreamControllerInvalidateBYOBRequest(*controller*).
2. Set *controller*.[[pendingPullIntos]] to a new empty List.

3.13.6. ReadableByteStreamControllerClose (*controller*) throws §

1. Let *stream* be *controller*.[[controlledReadableByteStream]].
2. Assert: *controller*.[[closeRequested]] is **false**.
3. Assert: *stream*.[[state]] is `"readable"`.
4. If *controller*.[[queueTotalSize]] > **0**,
    a. Set *controller*.[[closeRequested]] to **true**.
    b. Return.
5. If *controller*.[[pendingPullIntos]] is not empty,
    a. Let *firstPendingPullInto* be the first element of *controller*.[[pendingPullIntos]].
    b. If *firstPendingPullInto*.[[bytesFilled]] > **0**,
        i. Let *e* be a new **TypeError** exception.
        ii. Perform ! ReadableByteStreamControllerError(*controller*, *e*).
        iii. Throw *e*.
6. Perform ! ReadableByteStreamControllerClearAlgorithms(*controller*).
7. Perform ! ReadableStreamClose(*stream*).

3.13.7. ReadableByteStreamControllerCommitPullIntoDescriptor (*stream*, *pullIntoDescriptor*) nothrow §

1. Assert: *stream*.[[state]] is not `"errored"`.
2. Let *done* be **false**.
3. If *stream*.[[state]] is `"closed"`,
    a. Assert: *pullIntoDescriptor*.[[bytesFilled]] is **0**.
    b. Set *done* to **true**.
4. Let *filledView* be ! ReadableByteStreamControllerConvertPullIntoDescriptor(*pullIntoDescriptor*).
5. If *pullIntoDescriptor*.[[readerType]] is `"default"`,
    a. Perform ! ReadableStreamFulfillReadRequest(*stream*, *filledView*, *done*).
6. Otherwise,
    a. Assert: *pullIntoDescriptor*.[[readerType]] is `"byob"`.
    b. Perform ! ReadableStreamFulfillReadIntoRequest(*stream*, *filledView*, *done*).

3.13.8. ReadableByteStreamControllerConvertPullIntoDescriptor (*pullIntoDescriptor*) nothrow §

1. Let *bytesFilled* be *pullIntoDescriptor*.[[bytesFilled]].
2. Let *elementSize* be *pullIntoDescriptor*.[[elementSize]].
3. Assert: *bytesFilled* ≤ *pullIntoDescriptor*.[[byteLength]].
4. Assert: *bytesFilled* mod *elementSize* is **0**.
5. Return ! Construct(*pullIntoDescriptor*.[[ctor]], « *pullIntoDescriptor*.[[buffer]], *pullIntoDescriptor*.[[byteOffset]], *bytesFilled* ÷ *elementSize* »).

3.13.9. ReadableByteStreamControllerEnqueue (*controller*, *chunk*) nothrow §

1. Let *stream* be *controller*.[[controlledReadableByteStream]].
2. Assert: *controller*.[[closeRequested]] is **false**.
3. Assert: *stream*.[[state]] is `"readable"`.
4. Let *buffer* be *chunk*.[[ViewedArrayBuffer]].
5. Let *byteOffset* be *chunk*.[[ByteOffset]].
6. Let *byteLength* be *chunk*.[[ByteLength]].
7. Let *transferredBuffer* be ! TransferArrayBuffer(*buffer*).
8. If ! ReadableStreamHasDefaultReader(*stream*) is **true**
    a. If ! ReadableStreamGetNumReadRequests(*stream*) is **0**,
        i. Perform ! ReadableByteStreamControllerEnqueueChunkToQueue(*controller*, *transferredBuffer*, *byteOffset*, *byteLength*).
    b. Otherwise,
        i. Assert: *controller*.[[queue]] is empty.
        ii. Let *transferredView* be ! Construct(%Uint8Array%, « *transferredBuffer*, *byteOffset*, *byteLength* »).
        iii. Perform ! ReadableStreamFulfillReadRequest(*stream*, *transferredView*, **false**).
9. Otherwise, if ! ReadableStreamHasBYOBReader(*stream*) is **true**,
    a. Perform ! ReadableByteStreamControllerEnqueueChunkToQueue(*controller*, *transferredBuffer*, *byteOffset*, *byteLength*).
    b. Perform ! ReadableByteStreamControllerProcessPullIntoDescriptorsUsingQueue(*controller*).
10. Otherwise,
    a. Assert: ! IsReadableStreamLocked(*stream*) is **false**.
    b. Perform ! ReadableByteStreamControllerEnqueueChunkToQueue(*controller*, *transferredBuffer*, *byteOffset*, *byteLength*).
11. Perform ! ReadableByteStreamControllerCallPullIfNeeded(*controller*).

3.13.10. ReadableByteStreamControllerEnqueueChunkToQueue (*controller*, *buffer*, *byteOffset*, *byteLength*) nothrow §

1. Append Record {[[buffer]]: *buffer*, [[byteOffset]]: *byteOffset*, [[byteLength]]: *byteLength*} as the last element of *controller*.[[queue]].
2. Add *byteLength* to *controller*.[[queueTotalSize]].

3.13.11. ReadableByteStreamControllerError (*controller*, *e*) nothrow §

1. Let *stream* be *controller*.[[controlledReadableByteStream]].
2. If *stream*.[[state]] is not `"readable"`, return.
3. Perform ! ReadableByteStreamControllerClearPendingPullIntos(*controller*).
4. Perform ! ResetQueue(*controller*).
5. Perform ! ReadableByteStreamControllerClearAlgorithms(*controller*).
6. Perform ! ReadableStreamError(*stream*, *e*).

3.13.12. ReadableByteStreamControllerFillHeadPullIntoDescriptor (*controller*, *size*, *pullIntoDescriptor*) nothrow §

1. Assert: either *controller*.[[pendingPullIntos]] is empty, or the first element of *controller*.[[pendingPullIntos]] is *pullIntoDescriptor*.
2. Perform ! ReadableByteStreamControllerInvalidateBYOBRequest(*controller*).
3. Set *pullIntoDescriptor*.[[bytesFilled]] to *pullIntoDescriptor*.[[bytesFilled]] + *size*.

3.13.13. ReadableByteStreamControllerFillPullIntoDescriptorFromQueue (*controller*, *pullIntoDescriptor*) nothrow §

1. Let *elementSize* be *pullIntoDescriptor*.[[elementSize]].
2. Let *currentAlignedBytes* be *pullIntoDescriptor*.[[bytesFilled]] − (*pullIntoDescriptor*.[[bytesFilled]] mod *elementSize*).
3. Let *maxBytesToCopy* be min(*controller*.[[queueTotalSize]], *pullIntoDescriptor*.[[byteLength]] − *pullIntoDescriptor*.[[bytesFilled]]).
4. Let *maxBytesFilled* be *pullIntoDescriptor*.[[bytesFilled]] + *maxBytesToCopy*.
5. Let *maxAlignedBytes* be *maxBytesFilled* − (*maxBytesFilled* mod *elementSize*).
6. Let *totalBytesToCopyRemaining* be *maxBytesToCopy*.
7. Let *ready* be **false**.
8. If *maxAlignedBytes* > *currentAlignedBytes*,
    a. Set *totalBytesToCopyRemaining* to *maxAlignedBytes* − *pullIntoDescriptor*.[[bytesFilled]].
    b. Set *ready* to **true**.
9. Let *queue* be *controller*.[[queue]].
10. Repeat the following steps while *totalBytesToCopyRemaining* > **0**,
    a. Let *headOfQueue* be the first element of *queue*.
    b. Let *bytesToCopy* be min(*totalBytesToCopyRemaining*, *headOfQueue*.[[byteLength]]).
    c. Let *destStart* be *pullIntoDescriptor*.[[byteOffset]] + *pullIntoDescriptor*.[[bytesFilled]].
    d. Perform ! CopyDataBlockBytes(*pullIntoDescriptor*.[[buffer]].[[ArrayBufferData]], *destStart*, *headOfQueue*.[[buffer]].[[ArrayBufferData]], byteOffset]], *bytesToCopy*).

e. If *headOfQueue*.[[byteLength]] is *bytesToCopy*,
    i. Remove the first element of *queue*, shifting all other elements downward (so that the second becomes the first, and so on).
f. Otherwise,
    i. Set *headOfQueue*.[[byteOffset]] to *headOfQueue*.[[byteOffset]] + *bytesToCopy*.
    ii. Set *headOfQueue*.[[byteLength]] to *headOfQueue*.[[byteLength]] − *bytesToCopy*.
g. Set *controller*.[[queueTotalSize]] to *controller*.[[queueTotalSize]] − *bytesToCopy*.
h. Perform ! ReadableByteStreamControllerFillHeadPullIntoDescriptor(*controller*, *bytesToCopy*, *pullIntoDescriptor*).
i. Set *totalBytesToCopyRemaining* to *totalBytesToCopyRemaining* − *bytesToCopy*.

11. If *ready* is **false**,
    a. Assert: *controller*.[[queueTotalSize]] is **0**.
    b. Assert: *pullIntoDescriptor*.[[bytesFilled]] > **0**.
    c. Assert: *pullIntoDescriptor*.[[bytesFilled]] < *pullIntoDescriptor*.[[elementSize]].
12. Return *ready*.

3.13.14. ReadableByteStreamControllerGetDesiredSize (*controller*) nothrow §

1. Let *stream* be *controller*.[[controlledReadableByteStream]].
2. Let *state* be *stream*.[[state]].
3. If *state* is `"errored"`, return **null**.
4. If *state* is `"closed"`, return **0**.
5. Return *controller*.[[strategyHWM]] − *controller*.[[queueTotalSize]].

3.13.15. ReadableByteStreamControllerHandleQueueDrain (*controller*) nothrow §

1. Assert: *controller*.[[controlledReadableByteStream]].[[state]] is `"readable"`.
2. If *controller*.[[queueTotalSize]] is **0** and *controller*.[[closeRequested]] is **true**,
    a. Perform ! ReadableByteStreamControllerClearAlgorithms(*controller*).
    b. Perform ! ReadableStreamClose(*controller*.[[controlledReadableByteStream]]).
3. Otherwise,
    a. Perform ! ReadableByteStreamControllerCallPullIfNeeded(*controller*).

3.13.16. ReadableByteStreamControllerInvalidateBYOBRequest (*controller*) nothrow §

1. If *controller*.[[byobRequest]] is **undefined**, return.
2. Set *controller*.[[byobRequest]].[[associatedReadableByteStreamController]] to **undefined**.
3. Set *controller*.[[byobRequest]].[[view]] to **undefined**.
4. Set *controller*.[[byobRequest]] to **undefined**.

3.13.17. ReadableByteStreamControllerProcessPullIntoDescriptorsUsingQueue (*controller*) nothrow §

1. Assert: *controller*.[[closeRequested]] is **false**.
2. Repeat the following steps while *controller*.[[pendingPullIntos]] is not empty,
    a. If *controller*.[[queueTotalSize]] is **0**, return.
    b. Let *pullIntoDescriptor* be the first element of *controller*.[[pendingPullIntos]].
    c. If ! ReadableByteStreamControllerFillPullIntoDescriptorFromQueue(*controller*, *pullIntoDescriptor*) is **true**,
        i. Perform ! ReadableByteStreamControllerShiftPendingPullInto(*controller*).
        ii. Perform ! ReadableByteStreamControllerCommitPullIntoDescriptor(*controller*.[[controlledReadableByteStream]], *pullIntoDescriptor*).

3.13.18. ReadableByteStreamControllerPullInto (*controller*, *view*) nothrow §

1. Let *stream* be *controller*.[[controlledReadableByteStream]].
2. Let *elementSize* be 1.
3. Let *ctor* be %DataView%.
4. If *view* has a [[TypedArrayName]] internal slot (i.e., it is not a `DataView`),
    a. Set *elementSize* to the element size specified in the typed array constructors table for *view*.[[TypedArrayName]].
    b. Set *ctor* to the constructor specified in the typed array constructors table for *view*.[[TypedArrayName]].
5. Let *byteOffset* be *view*.[[ByteOffset]].

File an issue about the selected text

6. Let *byteLength* be *view*.[[ByteLength]].
7. Let *buffer* be ! TransferArrayBuffer(*view*.[[ViewedArrayBuffer]]).
8. Let *pullIntoDescriptor* be Record {[[buffer]]: *buffer*, [[byteOffset]]: *byteOffset*, [[byteLength]]: *byteLength*, [[bytesFilled]]: **0**, [[elementSize]]: *elementSize*, [[ctor]]: *ctor*, [[readerType]]: `"byob"`}.
9. If *controller*.[[pendingPullIntos]] is not empty,
    a. Append *pullIntoDescriptor* as the last element of *controller*.[[pendingPullIntos]].
    b. Return ! ReadableStreamAddReadIntoRequest(*stream*).
10. If *stream*.[[state]] is `"closed"`,
    a. Let *emptyView* be ! Construct(*ctor*, « *pullIntoDescriptor*.[[buffer]], *pullIntoDescriptor*.[[byteOffset]], **0** »).
    b. Return a promise resolved with ! ReadableStreamCreateReadResult(*emptyView*, **true**, *stream*.[[reader]].[[forAuthorCode]]).
11. If *controller*.[[queueTotalSize]] > **0**,
    a. If ! ReadableByteStreamControllerFillPullIntoDescriptorFromQueue(*controller*, *pullIntoDescriptor*) is **true**,
        i. Let *filledView* be ! ReadableByteStreamControllerConvertPullIntoDescriptor(*pullIntoDescriptor*).
        ii. Perform ! ReadableByteStreamControllerHandleQueueDrain(*controller*).
        iii. Return a promise resolved with ! ReadableStreamCreateReadResult(*filledView*, **false**, *stream*.[[reader]].[[forAuthorCode]]).
    b. If *controller*.[[closeRequested]] is **true**,
        i. Let *e* be a **TypeError** exception.
        ii. Perform ! ReadableByteStreamControllerError(*controller*, *e*).
        iii. Return a promise rejected with *e*.
12. Append *pullIntoDescriptor* as the last element of *controller*.[[pendingPullIntos]].
13. Let *promise* be ! ReadableStreamAddReadIntoRequest(*stream*).
14. Perform ! ReadableByteStreamControllerCallPullIfNeeded(*controller*).
15. Return *promise*.

3.13.19. ReadableByteStreamControllerRespond (*controller*, *bytesWritten*) throws §

1. Let *bytesWritten* be ? ToNumber(*bytesWritten*).
2. If ! IsFiniteNonNegativeNumber(*bytesWritten*) is **false**,
    a. Throw a **RangeError** exception.
3. Assert: *controller*.[[pendingPullIntos]] is not empty.
4. Perform ? ReadableByteStreamControllerRespondInternal(*controller*, *bytesWritten*).

3.13.20. ReadableByteStreamControllerRespondInClosedState (*controller*, *firstDescriptor*) nothrow §

1. Set *firstDescriptor*.[[buffer]] to ! TransferArrayBuffer(*firstDescriptor*.[[buffer]]).
2. Assert: *firstDescriptor*.[[bytesFilled]] is **0**.
3. Let *stream* be *controller*.[[controlledReadableByteStream]].
4. If ! ReadableStreamHasBYOBReader(*stream*) is **true**,
    a. Repeat the following steps while ! ReadableStreamGetNumReadIntoRequests(*stream*) > **0**,
        i. Let *pullIntoDescriptor* be ! ReadableByteStreamControllerShiftPendingPullInto(*controller*).
        ii. Perform ! ReadableByteStreamControllerCommitPullIntoDescriptor(*stream*, *pullIntoDescriptor*).

3.13.21. ReadableByteStreamControllerRespondInReadableState (*controller*, *bytesWritten*, *pullIntoDescriptor*) throws §

1. If *pullIntoDescriptor*.[[bytesFilled]] + *bytesWritten* > *pullIntoDescriptor*.[[byteLength]], throw a **RangeError** exception.
2. Perform ! ReadableByteStreamControllerFillHeadPullIntoDescriptor(*controller*, *bytesWritten*, *pullIntoDescriptor*).
3. If *pullIntoDescriptor*.[[bytesFilled]] < *pullIntoDescriptor*.[[elementSize]], return.
4. Perform ! ReadableByteStreamControllerShiftPendingPullInto(*controller*).
5. Let *remainderSize* be *pullIntoDescriptor*.[[bytesFilled]] mod *pullIntoDescriptor*.[[elementSize]].
6. If *remainderSize* > **0**,
    a. Let *end* be *pullIntoDescriptor*.[[byteOffset]] + *pullIntoDescriptor*.[[bytesFilled]].
    b. Let *remainder* be ? CloneArrayBuffer(*pullIntoDescriptor*.[[buffer]], *end* − *remainderSize*, *remainderSize*, %ArrayBuffer%).
    c. Perform ! ReadableByteStreamControllerEnqueueChunkToQueue(*controller*, *remainder*, **0**, *remainder*.[[ByteLength]]).
7. Set *pullIntoDescriptor*.[[buffer]] to ! TransferArrayBuffer(*pullIntoDescriptor*.[[buffer]]).
8. Set *pullIntoDescriptor*.[[bytesFilled]] to *pullIntoDescriptor*.[[bytesFilled]] − *remainderSize*.
9. Perform ! ReadableByteStreamControllerCommitPullIntoDescriptor(*controller*.[[controlledReadableByteStream]], *pullIntoDescriptor*).
10. Perform ! ReadableByteStreamControllerProcessPullIntoDescriptorsUsingQueue(*controller*).

3.13.22. ReadableByteStreamControllerRespondInternal (*controller*, *bytesWritten*) throws §

File an issue about the selected text first element of *controller*.[[pendingPullIntos]].

2. Let *stream* be *controller*.[[controlledReadableByteStream]].
3. If *stream*.[[state]] is `"closed"`,
    a. If *bytesWritten* is not **0**, throw a **TypeError** exception.
    b. Perform ! ReadableByteStreamControllerRespondInClosedState(*controller*, *firstDescriptor*).
4. Otherwise,
    a. Assert: *stream*.[[state]] is `"readable"`.
    b. Perform ? ReadableByteStreamControllerRespondInReadableState(*controller*, *bytesWritten*, *firstDescriptor*).
5. Perform ! ReadableByteStreamControllerCallPullIfNeeded(*controller*).

3.13.23. ReadableByteStreamControllerRespondWithNewView (*controller*, *view*) throws §

1. Assert: *controller*.[[pendingPullIntos]] is not empty.
2. Let *firstDescriptor* be the first element of *controller*.[[pendingPullIntos]].
3. If *firstDescriptor*.[[byteOffset]] + *firstDescriptor*.[[bytesFilled]] is not *view*.[[ByteOffset]], throw a **RangeError** exception.
4. If *firstDescriptor*.[[byteLength]] is not *view*.[[ByteLength]], throw a **RangeError** exception.
5. Set *firstDescriptor*.[[buffer]] to *view*.[[ViewedArrayBuffer]].
6. Perform ? ReadableByteStreamControllerRespondInternal(*controller*, *view*.[[ByteLength]]).

3.13.24. ReadableByteStreamControllerShiftPendingPullInto (*controller*) nothrow §

1. Let *descriptor* be the first element of *controller*.[[pendingPullIntos]].
2. Remove *descriptor* from *controller*.[[pendingPullIntos]], shifting all other elements downward (so that the second becomes the first, and so on).
3. Perform ! ReadableByteStreamControllerInvalidateBYOBRequest(*controller*).
4. Return *descriptor*.

3.13.25. ReadableByteStreamControllerShouldCallPull (*controller*) nothrow §

1. Let *stream* be *controller*.[[controlledReadableByteStream]].
2. If *stream*.[[state]] is not `"readable"`, return **false**.
3. If *controller*.[[closeRequested]] is **true**, return **false**.
4. If *controller*.[[started]] is **false**, return **false**.
5. If ! ReadableStreamHasDefaultReader(*stream*) is **true** and ! ReadableStreamGetNumReadRequests(*stream*) > **0**, return **true**.
6. If ! ReadableStreamHasBYOBReader(*stream*) is **true** and ! ReadableStreamGetNumReadIntoRequests(*stream*) > **0**, return **true**.
7. Let *desiredSize* be ! ReadableByteStreamControllerGetDesiredSize(*controller*).
8. Assert: *desiredSize* is not **null**.
9. If *desiredSize* > **0**, return **true**.
10. Return **false**.

3.13.26. SetUpReadableByteStreamController (*stream*, *controller*, *startAlgorithm*, *pullAlgorithm*, *cancelAlgorithm*, *highWaterMark*, *autoAllocateChunkSize*) throws §

1. Assert: *stream*.[[readableStreamController]] is **undefined**.
2. If *autoAllocateChunkSize* is not **undefined**,
    a. Assert: ! IsInteger(*autoAllocateChunkSize*) is **true**.
    b. Assert: *autoAllocateChunkSize* is positive.
3. Set *controller*.[[controlledReadableByteStream]] to *stream*.
4. Set *controller*.[[pullAgain]] and *controller*.[[pulling]] to **false**.
5. Set *controller*.[[byobRequest]] to **undefined**.
6. Perform ! ResetQueue(*controller*).
7. Set *controller*.[[closeRequested]] and *controller*.[[started]] to **false**.
8. Set *controller*.[[strategyHWM]] to ? ValidateAndNormalizeHighWaterMark(*highWaterMark*).
9. Set *controller*.[[pullAlgorithm]] to *pullAlgorithm*.
10. Set *controller*.[[cancelAlgorithm]] to *cancelAlgorithm*.
11. Set *controller*.[[autoAllocateChunkSize]] to *autoAllocateChunkSize*.
12. Set *controller*.[[pendingPullIntos]] to a new empty List.
13. Set *stream*.[[readableStreamController]] to *controller*.
14. Let *startResult* be the result of performing *startAlgorithm*.
15. Let *startPromise* be a promise resolved with *startResult*.
16. Upon fulfillment of *startPromise*,
    a. Set *controller*.[[started]] to **true**.

File an issue about the selected text r.[[pulling]] is **false**.

c. Assert: *controller*.[[pullAgain]] is **false**.
d. Perform ! ReadableByteStreamControllerCallPullIfNeeded(*controller*).

17. Upon rejection of *startPromise* with reason *r*,
    a. Perform ! ReadableByteStreamControllerError(*controller*, *r*).

3.13.27. SetUpReadableByteStreamControllerFromUnderlyingSource (*stream*, *underlyingByteSource*, *highWaterMark*) throws §

1. Assert: *underlyingByteSource* is not **undefined**.
2. Let *controller* be ObjectCreate(the original value of `ReadableByteStreamController`'s `prototype` property).
3. Let *startAlgorithm* be the following steps:
    a. Return ? InvokeOrNoop(*underlyingByteSource*, `"start"`, « *controller* »).
4. Let *pullAlgorithm* be ? CreateAlgorithmFromUnderlyingMethod(*underlyingByteSource*, `"pull"`, **0**, « *controller* »).
5. Let *cancelAlgorithm* be ? CreateAlgorithmFromUnderlyingMethod(*underlyingByteSource*, `"cancel"`, **1**, « »).
6. Let *autoAllocateChunkSize* be ? GetV(*underlyingByteSource*, `"autoAllocateChunkSize"`).
7. If *autoAllocateChunkSize* is not **undefined**,
    a. Set *autoAllocateChunkSize* to ? ToNumber(*autoAllocateChunkSize*).
    b. If ! IsInteger(*autoAllocateChunkSize*) is **false**, or if *autoAllocateChunkSize* ≤ **0**, throw a **RangeError** exception.
8. Perform ? SetUpReadableByteStreamController(*stream*, *controller*, *startAlgorithm*, *pullAlgorithm*, *cancelAlgorithm*, *highWaterMark*, *autoAllocateChunkSize*).

3.13.28. SetUpReadableStreamBYOBRequest (*request*, *controller*, *view*) nothrow §

1. Assert: ! IsReadableByteStreamController(*controller*) is **true**.
2. Assert: Type(*view*) is Object.
3. Assert: *view* has a [[ViewedArrayBuffer]] internal slot.
4. Assert: ! IsDetachedBuffer(*view*.[[ViewedArrayBuffer]]) is **false**.
5. Set *request*.[[associatedReadableByteStreamController]] to *controller*.
6. Set *request*.[[view]] to *view*.

4. Writable streams §

4.1. Using writable streams §

Example

The usual way to write to a writable stream is to simply pipe a readable stream to it. This ensures that backpressure is respected, so that if the writable stream's underlying sink is not able to accept data as fast as the readable stream can produce it, the readable stream is informed of this and has a chance to slow down its data production.

```
readableStream.pipeTo(writableStream)
  .then(() => console.log("All data successfully written!"))
  .catch(e => console.error("Something went wrong!", e));
```

Example

You can also write directly to writable streams by acquiring a writer and using its `write()` and `close()` methods. Since writable streams queue any incoming writes, and take care internally to forward them to the underlying sink in sequence, you can indiscriminately write to a writable stream without much ceremony:

```
function writeArrayToStream(array, writableStream) {
  const writer = writableStream.getWriter();
  array.forEach(chunk => writer.write(chunk).catch(() => {}));

  return writer.close();
}

writeArrayToStream([1, 2, 3, 4, 5], writableStream)
  .then(() => console.log("All done!"))
  .catch(e => console.error("Error with the stream: " + e));
```

Note how we use `.catch(() => {})` to suppress any rejections from the `write()` method; we'll be notified of any fatal errors via a rejection of the `close()` method, and leaving them un-caught would cause potential `unhandledrejection` events and console warnings.

Example

In the previous example we only paid attention to the success or failure of the entire stream, by looking at the promise returned by the writer's `close()` method. That promise will reject if anything goes wrong with the stream—initializing it, writing to it, or closing it. And it will fulfill once the stream is successfully closed. Often this is all you care about.

However, if you care about the success of writing a specific chunk, you can use the promise returned by the writer's `write()` method:

```
writer.write("i am a chunk of data")
  .then(() => console.log("chunk successfully written!"))
  .catch(e => console.error(e));
```

What "success" means is up to a given stream instance (or more precisely, its underlying sink) to decide. For example, for a file stream it could simply mean that the OS has accepted the write, and not necessarily that the chunk has been flushed to disk. Some streams might not be able to give such a signal at all, in which case the returned promise will fulfill immediately.

Example

The `desiredSize` and `ready` properties of writable stream writers allow producers to more precisely respond to flow control signals from the stream, to keep memory usage below the stream's specified high water mark. The following example writes an infinite sequence of random bytes to a stream, using `desiredSize` to determine how many bytes to generate at a given time, and using `ready` to wait for the backpressure to subside.

```
async function writeRandomBytesForever(writableStream) {
  const writer = writableStream.getWriter();

  while (true) {
    await writer.ready;

    const bytes = new Uint8Array(writer.desiredSize);
    domValues(bytes);
```

```
      // Purposefully don't await; awaiting writer.ready is enough.
      writer.write(bytes).catch(() => {});
    }
  }

  writeRandomBytesForever(myWritableStream).catch(e => console.error("Something broke", e));
```

Note how we don't `await` the promise returned by `write()`; this would be redundant with `await`ing the `ready` promise. Additionally, similar to a previous example, we use the `.catch(() => {})` pattern on the promises returned by `write()`; in this case we'll be notified about any failures `await`ing the `ready` promise.

Example

To further emphasize how it's a bad idea to `await` the promise returned by `write()`, consider a modification of the above example, where we continue to use the `WritableStreamDefaultWriter` interface directly, but we don't control how many bytes we have to write at a given time. In that case, the backpressure-respecting code looks the same:

```
async function writeSuppliedBytesForever(writableStream, getBytes) {
  const writer = writableStream.getWriter();

  while (true) {
    await writer.ready;

    const bytes = getBytes();
    writer.write(bytes).catch(() => {});
  }
}
```

Unlike the previous example, where—because we were always writing exactly `writer.desiredSize` bytes each time—the `write()` and `ready` promises were synchronized, in this case it's quite possible that the `ready` promise fulfills before the one returned by `write()` does. Remember, the `ready` promise fulfills when the desired size becomes positive, which might be before the write succeeds (especially in cases with a larger high water mark).

In other words, `await`ing the return value of `write()` means you never queue up writes in the stream's internal queue, instead only executing a write after the previous one succeeds, which can result in low throughput.

4.2. Class `WritableStream`

4.2.1. Class definition §

This section is non-normative.

If one were to write the `WritableStream` class in something close to the syntax of [ECMASCRIPT], it would look like

```
class WritableStream {
  constructor(underlyingSink = {}, strategy = {})

  get locked()

  abort(reason)
  getWriter()
}
```

4.2.2. Internal slots §

Instances of `WritableStream` are created with the internal slots described in the following table:

Internal Slot	Description (*non-normative*)
[[backpressure]]	The backpressure signal set by the controller
[[closeRequest]]	The promise returned from the writer `close()` method
[[inFlightWriteRequest]]	A slot set to the promise for the current in-flight write operation while the underlying sink's write algorithm is executing and has not yet fulfilled, used to entrant calls

File an issue about the selected text

Internal Slot	Description (*non-normative*)
[[inFlightCloseRequest]]	A slot set to the promise for the current in-flight close operation while the underlying sink's close algorithm is executing and has not yet fulfilled, used to prevent the `abort()` method from interrupting close
[[pendingAbortRequest]]	A Record containing the promise returned from `abort()` and the *reason* passed to `abort()`
[[state]]	A string containing the stream's current state, used internally; one of `"writable"`, `"closed"`, `"erroring"`, or `"errored"`
[[storedError]]	A value indicating how the stream failed, to be given as a failure reason or exception when trying to operate on the stream while in the `"errored"` state
[[writableStreamController]]	A `WritableStreamDefaultController` created with the ability to control the state and queue of this stream; also used for the IsWritableStream brand check
[[writer]]	A `WritableStreamDefaultWriter` instance, if the stream is locked to a writer, or **undefined** if it is not
[[writeRequests]]	A List of promises representing the stream's internal queue of write requests not yet processed by the underlying sink

Note

*The [[inFlightCloseRequest]] slot and [[closeRequest]] slot are mutually exclusive. Similarly, no element will be removed from [[writeRequests]] while [[inFlightWriteRequest]] is not **undefined**. Implementations can optimize storage for these slots based on these invariants.*

4.2.3. new WritableStream(*underlyingSink* = {}, *strategy* = {})

Note

The `underlyingSink` argument represents the underlying sink, as described in §4.2.4 Underlying sink API.

The `strategy` argument represents the stream's queuing strategy, as described in §6.1.1 The queuing strategy API. If it is not provided, the default behavior will be the same as a `CountQueuingStrategy` with a high water mark of 1.

1. Perform ! InitializeWritableStream(*this*).
2. Let *size* be ? GetV(*strategy*, `"size"`).
3. Let *highWaterMark* be ? GetV(*strategy*, `"highWaterMark"`).
4. Let *type* be ? GetV(*underlyingSink*, `"type"`).
5. If *type* is not **undefined**, throw a **RangeError** exception.

   Note

   This is to allow us to add new potential types in the future, without backward-compatibility concerns.

6. Let *sizeAlgorithm* be ? MakeSizeAlgorithmFromSizeFunction(*size*).
7. If *highWaterMark* is **undefined**, let *highWaterMark* be **1**.
8. Set *highWaterMark* to ? ValidateAndNormalizeHighWaterMark(*highWaterMark*).
9. Perform ? SetUpWritableStreamDefaultControllerFromUnderlyingSink(**this**, *underlyingSink*, *highWaterMark*, *sizeAlgorithm*).

4.2.4. Underlying sink API §

This section is non-normative.

The `WritableStream()` constructor accepts as its first argument a JavaScript object representing the underlying sink. Such objects can contain any of the following properties:

`start(controller)`

A function that is called immediately during creation of the `WritableStream`.

Typically this is used to acquire access to the underlying sink resource being represented.

If this setup process is asynchronous, it can return a promise to signal success or failure; a rejected promise will error the stream. Any thrown exceptions will be re-thrown by the `WritableStream()` constructor.

`write(chunk, controller)`

A function that is called when a new chunk of data is ready to be written to the underlying sink. The stream implementation guarantees that this function will be called only after previous writes have succeeded, and never before `start()` has succeeded or after `close()` or `abort()` have been called.

This function is used to actually send the data to the resource presented by the underlying sink, for example by calling a lower-level API.

If the process of writing data is asynchronous, and communicates success or failure signals back to its user, then this function can return a promise to signal success or failure. This promise return value will be communicated back to the caller of `writer.write()`, so they can monitor that individual write. Throwing an exception is treated the same as returning a rejected promise.

File an issue about the selected text

Note that such signals are not always available; compare e.g. §8.6 A writable stream with no backpressure or success signals with §8.7 A writable stream with backpressure and success signals. In such cases, it's best to not return anything.

The promise potentially returned by this function also governs whether the given chunk counts as written for the purposes of computed the desired size to fill the stream's internal queue. That is, during the time it takes the promise to settle, `writer.desiredSize` will stay at its previous value, only increasing to signal the desire for more chunks once the write succeeds.

`close()`

A function that is called after the producer signals, via `writer.close()`, that they are done writing chunks to the stream, and subsequently all queued-up writes have successfully completed.

This function can perform any actions necessary to finalize or flush writes to the underlying sink, and release access to any held resources.

If the shutdown process is asynchronous, the function can return a promise to signal success or failure; the result will be communicated via the return value of the called `writer.close()` method. Additionally, a rejected promise will error the stream, instead of letting it close successfully. Throwing an exception is treated the same as returning a rejected promise.

`abort(reason)`

A function that is called after the producer signals, via `stream.abort()` or `writer.abort()`, that they wish to abort the stream. It takes as its argument the same value as was passed to those methods by the producer.

Writable streams can additionally be aborted under certain conditions during piping; see the definition of the `pipeTo()` method for more details.

This function can clean up any held resources, much like `close()`, but perhaps with some custom handling.

If the shutdown process is asynchronous, the function can return a promise to signal success or failure; the result will be communicated via the return value of the called `abort()` method. Throwing an exception is treated the same as returning a rejected promise. Regardless, the stream will be errored with a new `TypeError` indicating that it was aborted.

The `controller` argument passed to `start()` and `write()` is an instance of `WritableStreamDefaultController`, and has the ability to error the stream. This is mainly used for bridging the gap with non-promise-based APIs, as seen for example in §8.6 A writable stream with no backpressure or success signals.

4.2.5. Properties of the `WritableStream` prototype §

4.2.5.1. get locked

Note

The `locked` getter returns whether or not the writable stream is locked to a writer.

1. If ! IsWritableStream(**this**) is **false**, throw a **TypeError** exception.
2. Return ! IsWritableStreamLocked(**this**).

4.2.5.2. abort(*reason*)

Note

The `abort` method aborts the stream, signaling that the producer can no longer successfully write to the stream and it is to be immediately moved to an errored state, with any queued-up writes discarded. This will also execute any abort mechanism of the underlying sink.

1. If ! IsWritableStream(**this**) is **false**, return a promise rejected with a **TypeError** exception.
2. If ! IsWritableStreamLocked(**this**) is **true**, return a promise rejected with a **TypeError** exception.
3. Return ! WritableStreamAbort(**this**, *reason*).

4.2.5.3. getWriter()

Note

The `getWriter` method creates a writer (an instance of `WritableStreamDefaultWriter`) and locks the stream to the new writer. While the stream is locked, no other writer can be acquired until this one is released.

This functionality is especially useful for creating abstractions that desire the ability to write to a stream without interruption or interleaving. By getting a writer for the stream, you can ensure nobody else can write at the same time, which would cause the resulting written data to be unpredictable and probably useless.

File an issue about the selected text s) is **false**, throw a **TypeError** exception.

2. Return ? AcquireWritableStreamDefaultWriter(**this**).

4.3. General writable stream abstract operations §

The following abstract operations, unlike most in this specification, are meant to be generally useful by other specifications, instead of just being part of the implementation of this spec's classes.

4.3.1. AcquireWritableStreamDefaultWriter (*stream*) throws §

1. Return ? Construct(`WritableStreamDefaultWriter`, « *stream* »).

4.3.2. CreateWritableStream (*startAlgorithm*, *writeAlgorithm*, *closeAlgorithm*, *abortAlgorithm* [, *highWaterMark* [, *sizeAlgorithm*]]) throws §

This abstract operation is meant to be called from other specifications that wish to create `WritableStream` instances. The *writeAlgorithm*, *closeAlgorithm* and *abortAlgorithm* algorithms must return promises; if supplied, *sizeAlgorithm* must be an algorithm accepting chunk objects and returning a number; and if supplied, *highWaterMark* must be a non-negative, non-NaN number.

Note

CreateWritableStream throws an exception if and only if the supplied startAlgorithm *throws.*

1. If *highWaterMark* was not passed, set it to **1**.
2. If *sizeAlgorithm* was not passed, set it to an algorithm that returns **1**.
3. Assert: ! IsNonNegativeNumber(*highWaterMark*) is **true**.
4. Let *stream* be ObjectCreate(the original value of `WritableStream`'s `prototype` property).
5. Perform ! InitializeWritableStream(*stream*).
6. Let *controller* be ObjectCreate(the original value of `WritableStreamDefaultController`'s `prototype` property).
7. Perform ? SetUpWritableStreamDefaultController(*stream*, *controller*, *startAlgorithm*, *writeAlgorithm*, *closeAlgorithm*, *abortAlgorithm*, *highWaterMark*, *sizeAlgorithm*).
8. Return *stream*.

4.3.3. InitializeWritableStream (*stream*) nothrow §

1. Set *stream*.[[state]] to `"writable"`.
2. Set *stream*.[[storedError]], *stream*.[[writer]], *stream*.[[writableStreamController]], *stream*.[[inFlightWriteRequest]], *stream*.[[closeRequest]], *stream*.[[inFlightCloseRequest]] and *stream*.[[pendingAbortRequest]] to **undefined**.
3. Set *stream*.[[writeRequests]] to a new empty List.
4. Set *stream*.[[backpressure]] to **false**.

4.3.4. IsWritableStream (*x*) nothrow §

1. If Type(*x*) is not Object, return **false**.
2. If *x* does not have a [[writableStreamController]] internal slot, return **false**.
3. Return **true**.

4.3.5. IsWritableStreamLocked (*stream*) nothrow §

This abstract operation is meant to be called from other specifications that may wish to query whether or not a writable stream is locked to a writer.

1. Assert: ! IsWritableStream(*stream*) is **true**.
2. If *stream*.[[writer]] is **undefined**, return **false**.
3. Return **true**.

4.3.6. WritableStreamAbort (*stream*, *reason*) nothrow §

1. Let *state* be *stream*.[[state]].
2. If *state* is `"closed"` or `"errored"`, return a promise resolved with **undefined**.
3. If *stream*.[[pendingAbortRequest]] is not **undefined**, return *stream*.[[pendingAbortRequest]].[[promise]].
4. Assert: *state* is `"writable"` or `"erroring"`.
5. Let *wasAlreadyErroring* be **false**.
6. If *state* is `"erroring"`,
    a. Set *wasAlreadyErroring* to **true**.
    b. Set *reason* to **undefined**.
7. Let *promise* be a new promise.
8. Set *stream*.[[pendingAbortRequest]] to Record {[[promise]]: *promise*, [[reason]]: *reason*, [[wasAlreadyErroring]]: *wasAlreadyErroring*}.
9. If *wasAlreadyErroring* is **false**, perform ! WritableStreamStartErroring(*stream*, *reason*).
10. Return *promise*.

4.4. Writable stream abstract operations used by controllers §

To allow future flexibility to add different writable stream behaviors (similar to the distinction between default readable streams and readable byte streams), much of the internal state of a writable stream is encapsulated by the `WritableStreamDefaultController` class.

The abstract operations in this section are interfaces that are used by the controller implementation to affect its associated `WritableStream` object, translating the controller's internal state changes into developer-facing results visible through the `WritableStream`'s public API.

4.4.1. WritableStreamAddWriteRequest (*stream*) nothrow §

1. Assert: ! IsWritableStreamLocked(*stream*) is **true**.
2. Assert: *stream*.[[state]] is `"writable"`.
3. Let *promise* be a new promise.
4. Append *promise* as the last element of *stream*.[[writeRequests]].
5. Return *promise*.

4.4.2. WritableStreamDealWithRejection (*stream*, *error*) nothrow §

1. Let *state* be *stream*.[[state]].
2. If *state* is `"writable"`,
    a. Perform ! WritableStreamStartErroring(*stream*, *error*).
    b. Return.
3. Assert: *state* is `"erroring"`.
4. Perform ! WritableStreamFinishErroring(*stream*).

4.4.3. WritableStreamStartErroring (*stream*, *reason*) nothrow §

1. Assert: *stream*.[[storedError]] is **undefined**.
2. Assert: *stream*.[[state]] is `"writable"`.
3. Let *controller* be *stream*.[[writableStreamController]].
4. Assert: *controller* is not **undefined**.
5. Set *stream*.[[state]] to `"erroring"`.
6. Set *stream*.[[storedError]] to *reason*.
7. Let *writer* be *stream*.[[writer]].
8. If *writer* is not **undefined**, perform ! WritableStreamDefaultWriterEnsureReadyPromiseRejected(*writer*, *reason*).
9. If ! WritableStreamHasOperationMarkedInFlight(*stream*) is **false** and *controller*.[[started]] is **true**, perform ! WritableStreamFinishErroring(*stream*).

4.4.4. WritableStreamFinishErroring (*stream*) nothrow §

1. Assert: *stream*.[[state]] is `"erroring"`.
2. Assert: ! WritableStreamHasOperationMarkedInFlight(*stream*) is **false**.
3. Set *stream*.[[state]] to `"errored"`.

4. Perform ! *stream*.[[writableStreamController]].[[ErrorSteps]]().
5. Let *storedError* be *stream*.[[storedError]].
6. Repeat for each *writeRequest* that is an element of *stream*.[[writeRequests]],
    a. Reject *writeRequest* with *storedError*.
7. Set *stream*.[[writeRequests]] to an empty List.
8. If *stream*.[[pendingAbortRequest]] is **undefined**,
    a. Perform ! WritableStreamRejectCloseAndClosedPromiseIfNeeded(*stream*).
    b. Return.
9. Let *abortRequest* be *stream*.[[pendingAbortRequest]].
10. Set *stream*.[[pendingAbortRequest]] to **undefined**.
11. If *abortRequest*.[[wasAlreadyErroring]] is **true**,
    a. Reject *abortRequest*.[[promise]] with *storedError*.
    b. Perform ! WritableStreamRejectCloseAndClosedPromiseIfNeeded(*stream*).
    c. Return.
12. Let *promise* be ! stream.[[writableStreamController]].[[AbortSteps]](*abortRequest*.[[reason]]).
13. Upon fulfillment of *promise*,
    a. Resolve *abortRequest*.[[promise]] with **undefined**.
    b. Perform ! WritableStreamRejectCloseAndClosedPromiseIfNeeded(*stream*).
14. Upon rejection of *promise* with reason *reason*,
    a. Reject *abortRequest*.[[promise]] with *reason*.
    b. Perform ! WritableStreamRejectCloseAndClosedPromiseIfNeeded(*stream*).

4.4.5. WritableStreamFinishInFlightWrite (*stream*) nothrow §

1. Assert: *stream*.[[inFlightWriteRequest]] is not **undefined**.
2. Resolve *stream*.[[inFlightWriteRequest]] with **undefined**.
3. Set *stream*.[[inFlightWriteRequest]] to **undefined**.

4.4.6. WritableStreamFinishInFlightWriteWithError (*stream*, *error*) nothrow §

1. Assert: *stream*.[[inFlightWriteRequest]] is not **undefined**.
2. Reject *stream*.[[inFlightWriteRequest]] with *error*.
3. Set *stream*.[[inFlightWriteRequest]] to **undefined**.
4. Assert: *stream*.[[state]] is `"writable"` or `"erroring"`.
5. Perform ! WritableStreamDealWithRejection(*stream*, *error*).

4.4.7. WritableStreamFinishInFlightClose (*stream*) nothrow §

1. Assert: *stream*.[[inFlightCloseRequest]] is not **undefined**.
2. Resolve *stream*.[[inFlightCloseRequest]] with **undefined**.
3. Set *stream*.[[inFlightCloseRequest]] to **undefined**.
4. Let *state* be *stream*.[[state]].
5. Assert: *stream*.[[state]] is `"writable"` or `"erroring"`.
6. If *state* is `"erroring"`,
    a. Set *stream*.[[storedError]] to **undefined**.
    b. If *stream*.[[pendingAbortRequest]] is not **undefined**,
        i. Resolve *stream*.[[pendingAbortRequest]].[[promise]] with **undefined**.
        ii. Set *stream*.[[pendingAbortRequest]] to **undefined**.
7. Set *stream*.[[state]] to `"closed"`.
8. Let *writer* be *stream*.[[writer]].
9. If *writer* is not **undefined**, resolve *writer*.[[closedPromise]] with **undefined**.
10. Assert: *stream*.[[pendingAbortRequest]] is **undefined**.
11. Assert: *stream*.[[storedError]] is **undefined**.

4.4.8. WritableStreamFinishInFlightCloseWithError (*stream*, *error*) nothrow §

1. Assert: *stream*.[[inFlightCloseRequest]] is not **undefined**.
2. Reject *stream*.[[inFlightCloseRequest]] with *error*.
3. Set *stream*.[[inFlightCloseRequest]] to **undefined**.
4. Assert: *stream*.[[state]] is `"writable"` or `"erroring"`.

File an issue about the selected text Request]] is not **undefined**,

a. Reject *stream*.[[pendingAbortRequest]].[[promise]] with *error*.
b. Set *stream*.[[pendingAbortRequest]] to **undefined**.

6. Perform ! WritableStreamDealWithRejection(*stream*, *error*).

4.4.9. WritableStreamCloseQueuedOrInFlight (*stream*) nothrow §

1. If *stream*.[[closeRequest]] is **undefined** and *stream*.[[inFlightCloseRequest]] is **undefined**, return **false**.
2. Return **true**.

4.4.10. WritableStreamHasOperationMarkedInFlight (*stream*) nothrow §

1. If *stream*.[[inFlightWriteRequest]] is **undefined** and *controller*.[[inFlightCloseRequest]] is **undefined**, return **false**.
2. Return **true**.

4.4.11. WritableStreamMarkCloseRequestInFlight (*stream*) nothrow §

1. Assert: *stream*.[[inFlightCloseRequest]] is **undefined**.
2. Assert: *stream*.[[closeRequest]] is not **undefined**.
3. Set *stream*.[[inFlightCloseRequest]] to *stream*.[[closeRequest]].
4. Set *stream*.[[closeRequest]] to **undefined**.

4.4.12. WritableStreamMarkFirstWriteRequestInFlight (*stream*) nothrow §

1. Assert: *stream*.[[inFlightWriteRequest]] is **undefined**.
2. Assert: *stream*.[[writeRequests]] is not empty.
3. Let *writeRequest* be the first element of *stream*.[[writeRequests]].
4. Remove *writeRequest* from *stream*.[[writeRequests]], shifting all other elements downward (so that the second becomes the first, and so on).
5. Set *stream*.[[inFlightWriteRequest]] to *writeRequest*.

4.4.13. WritableStreamRejectCloseAndClosedPromiseIfNeeded (*stream*) nothrow §

1. Assert: *stream*.[[state]] is `"errored"`.
2. If *stream*.[[closeRequest]] is not **undefined**,
    a. Assert: *stream*.[[inFlightCloseRequest]] is **undefined**.
    b. Reject *stream*.[[closeRequest]] with *stream*.[[storedError]].
    c. Set *stream*.[[closeRequest]] to **undefined**.
3. Let *writer* be *stream*.[[writer]].
4. If *writer* is not **undefined**,
    a. Reject *writer*.[[closedPromise]] with *stream*.[[storedError]].
    b. Set *writer*.[[closedPromise]].[[PromiseIsHandled]] to **true**.

4.4.14. WritableStreamUpdateBackpressure (*stream*, *backpressure*) nothrow §

1. Assert: *stream*.[[state]] is `"writable"`.
2. Assert: ! WritableStreamCloseQueuedOrInFlight(*stream*) is **false**.
3. Let *writer* be *stream*.[[writer]].
4. If *writer* is not **undefined** and *backpressure* is not *stream*.[[backpressure]],
    a. If *backpressure* is **true**, set *writer*.[[readyPromise]] to a new promise.
    b. Otherwise,
        i. Assert: *backpressure* is **false**.
        ii. Resolve *writer*.[[readyPromise]] with **undefined**.
5. Set *stream*.[[backpressure]] to *backpressure*.

4.5. Class `WritableStreamDefaultWriter`

The `WritableStreamDefaultWriter` class represents a writable stream writer designed to be vended by a `WritableStream` instance.

4.5.1. Class definition §

This section is non-normative.

If one were to write the `WritableStreamDefaultWriter` class in something close to the syntax of [ECMASCRIPT], it would look like

```
class WritableStreamDefaultWriter {
  constructor(stream)

  get closed()
  get desiredSize()
  get ready()

  abort(reason)
  close()
  releaseLock()
  write(chunk)
}
```

4.5.2. Internal slots §

Instances of `WritableStreamDefaultWriter` are created with the internal slots described in the following table:

Internal Slot	Description (*non-normative*)
[[closedPromise]]	A promise returned by the writer's `closed` getter
[[ownerWritableStream]]	A `WritableStream` instance that owns this writer
[[readyPromise]]	A promise returned by the writer's `ready` getter

4.5.3. new WritableStreamDefaultWriter(*stream*)

Note

The `WritableStreamDefaultWriter` constructor is generally not meant to be used directly; instead, a stream's `getWriter()` method ought to be used.

1. If ! IsWritableStream(*stream*) is **false**, throw a **TypeError** exception.
2. If ! IsWritableStreamLocked(*stream*) is **true**, throw a **TypeError** exception.
3. Set **this**.[[ownerWritableStream]] to *stream*.
4. Set *stream*.[[writer]] to **this**.
5. Let *state* be *stream*.[[state]].
6. If *state* is `"writable"`,
    a. If ! WritableStreamCloseQueuedOrInFlight(*stream*) is **false** and *stream*.[[backpressure]] is **true**, set **this**.[[readyPromise]] to a new promise.
    b. Otherwise, set **this**.[[readyPromise]] to a promise resolved with **undefined**.
    c. Set **this**.[[closedPromise]] to a new promise.
7. Otherwise, if *state* is `"erroring"`,
    a. Set **this**.[[readyPromise]] to a promise rejected with *stream*.[[storedError]].
    b. Set **this**.[[readyPromise]].[[PromiseIsHandled]] to **true**.
    c. Set **this**.[[closedPromise]] to a new promise.
8. Otherwise, if *state* is `"closed"`,
    a. Set **this**.[[readyPromise]] to a promise resolved with **undefined**.
    b. Set **this**.[[closedPromise]] to a promise resolved with **undefined**.
9. Otherwise,
    a. Assert: *state* is `"errored"`.
    b. Let *storedError* be *stream*.[[storedError]].
    c. Set **this**.[[readyPromise]] to a promise rejected with *storedError*.
    d. Set **this**.[[readyPromise]].[[PromiseIsHandled]] to **true**.
    e. Set **this**.[[closedPromise]] to a promise rejected with *storedError*.
    f. Set **this**.[[closedPromise]].[[PromiseIsHandled]] to **true**.

File an issue about the selected text

4.5.4. Properties of the `WritableStreamDefaultWriter` prototype §

4.5.4.1. get closed

Note

The `closed` getter returns a promise that will be fulfilled when the stream becomes closed, or rejected if the stream ever errors or the writer's lock is released before the stream finishes closing.

1. If ! IsWritableStreamDefaultWriter(**this**) is **false**, return a promise rejected with a **TypeError** exception.
2. Return **this**.[[closedPromise]].

4.5.4.2. get desiredSize

Note

The `desiredSize` getter returns the desired size to fill the stream's internal queue. It can be negative, if the queue is over-full. A producer can use this information to determine the right amount of data to write.

*It will be **null** if the stream cannot be successfully written to (due to either being errored, or having an abort queued up). It will return zero if the stream is closed. The getter will throw an exception if invoked when the writer's lock is released.*

1. If ! IsWritableStreamDefaultWriter(**this**) is **false**, throw a **TypeError** exception.
2. If **this**.[[ownerWritableStream]] is **undefined**, throw a **TypeError** exception.
3. Return ! WritableStreamDefaultWriterGetDesiredSize(**this**).

4.5.4.3. get ready

Note

The `ready` getter returns a promise that will be fulfilled when the desired size to fill the stream's internal queue transitions from non-positive to positive, signaling that it is no longer applying backpressure. Once the desired size to fill the stream's internal queue dips back to zero or below, the getter will return a new promise that stays pending until the next transition.

If the stream becomes errored or aborted, or the writer's lock is released, the returned promise will become rejected.

1. If ! IsWritableStreamDefaultWriter(**this**) is **false**, return a promise rejected with a **TypeError** exception.
2. Return **this**.[[readyPromise]].

4.5.4.4. abort(*reason*)

Note

If the writer is active, the `abort` method behaves the same as that for the associated stream. (Otherwise, it returns a rejected promise.)

1. If ! IsWritableStreamDefaultWriter(**this**) is **false**, return a promise rejected with a **TypeError** exception.
2. If **this**.[[ownerWritableStream]] is **undefined**, return a promise rejected with a **TypeError** exception.
3. Return ! WritableStreamDefaultWriterAbort(**this**, *reason*).

4.5.4.5. close()

Note

The `close` method will close the associated writable stream. The underlying sink will finish processing any previously-written chunks, before invoking its close behavior. During this time any further attempts to write will fail (without erroring the stream).

*The method returns a promise that is fulfilled with **undefined** if all remaining chunks are successfully written and the stream successfully closes, or rejects if an error is encountered during this process.*

1. If ! IsWritableStreamDefaultWriter(**this**) is **false**, return a promise rejected with a **TypeError** exception.
2. Let *stream* be **this**.[[ownerWritableStream]].
3. If *stream* is **undefined**, return a promise rejected with a **TypeError** exception.
4. If ! WritableStreamCloseQueuedOrInFlight(*stream*) is **true**, return a promise rejected with a **TypeError** exception.
5. Return ! WritableStreamDefaultWriterClose(**this**).

File an issue about the selected text

4.5.4.6. releaseLock()

Note

The `releaseLock` method releases the writer's lock on the corresponding stream. After the lock is released, the writer is no longer active. If the associated stream is errored when the lock is released, the writer will appear errored in the same way from now on; otherwise, the writer will appear closed.

Note that the lock can still be released even if some ongoing writes have not yet finished (i.e. even if the promises returned from previous calls to `write()` have not yet settled). It's not necessary to hold the lock on the writer for the duration of the write; the lock instead simply prevents other producers from writing in an interleaved manner.

1. If ! IsWritableStreamDefaultWriter(**this**) is **false**, throw a **TypeError** exception.
2. Let *stream* be **this**.[[ownerWritableStream]].
3. If *stream* is **undefined**, return.
4. Assert: *stream*.[[writer]] is not **undefined**.
5. Perform ! WritableStreamDefaultWriterRelease(**this**).

4.5.4.7. write(*chunk*)

Note

*The `write` method writes the given chunk to the writable stream, by waiting until any previous writes have finished successfully, and then sending the chunk to the underlying sink's `write()` method. It will return a promise that fulfills with **undefined** upon a successful write, or rejects if the write fails or stream becomes errored before the writing process is initiated.*

Note that what "success" means is up to the underlying sink; it might indicate simply that the chunk has been accepted, and not necessarily that it is safely saved to its ultimate destination.

1. If ! IsWritableStreamDefaultWriter(**this**) is **false**, return a promise rejected with a **TypeError** exception.
2. If **this**.[[ownerWritableStream]] is **undefined**, return a promise rejected with a **TypeError** exception.
3. Return ! WritableStreamDefaultWriterWrite(**this**, *chunk*).

4.6. Writable stream writer abstract operations §

4.6.1. IsWritableStreamDefaultWriter (*x*) nothrow §

1. If Type(*x*) is not Object, return **false**.
2. If *x* does not have an [[ownerWritableStream]] internal slot, return **false**.
3. Return **true**.

4.6.2. WritableStreamDefaultWriterAbort (*writer*, *reason*) nothrow §

1. Let *stream* be *writer*.[[ownerWritableStream]].
2. Assert: *stream* is not **undefined**.
3. Return ! WritableStreamAbort(*stream*, *reason*).

4.6.3. WritableStreamDefaultWriterClose (*writer*) nothrow §

1. Let *stream* be *writer*.[[ownerWritableStream]].
2. Assert: *stream* is not **undefined**.
3. Let *state* be *stream*.[[state]].
4. If *state* is `"closed"` or `"errored"`, return a promise rejected with a **TypeError** exception.
5. Assert: *state* is `"writable"` or `"erroring"`.
6. Assert: ! WritableStreamCloseQueuedOrInFlight(*stream*) is **false**.
7. Let *promise* be a new promise.
8. Set *stream*.[[closeRequest]] to *promise*.
9. If *stream*.[[backpressure]] is **true** and *state* is `"writable"`, resolve *writer*.[[readyPromise]] with **undefined**.
10. Perform ! WritableStreamDefaultControllerClose(*stream*.[[writableStreamController]]).
11. Return *promise*.

File an issue about the selected text

4.6.4. WritableStreamDefaultWriterCloseWithErrorPropagation (*writer*) nothrow §

Note

This abstract operation helps implement the error propagation semantics of `pipeTo()`.

1. Let *stream* be *writer*.[[ownerWritableStream]].
2. Assert: *stream* is not **undefined**.
3. Let *state* be *stream*.[[state]].
4. If ! WritableStreamCloseQueuedOrInFlight(*stream*) is **true** or *state* is `"closed"`, return a promise resolved with **undefined**.
5. If *state* is `"errored"`, return a promise rejected with *stream*.[[storedError]].
6. Assert: *state* is `"writable"` or `"erroring"`.
7. Return ! WritableStreamDefaultWriterClose(*writer*).

4.6.5. WritableStreamDefaultWriterEnsureClosedPromiseRejected(*writer*, *error*) nothrow §

1. If *writer*.[[closedPromise]].[[PromiseState]] is `"pending"`, reject *writer*.[[closedPromise]] with *error*.
2. Otherwise, set *writer*.[[closedPromise]] to a promise rejected with *error*.
3. Set *writer*.[[closedPromise]].[[PromiseIsHandled]] to **true**.

4.6.6. WritableStreamDefaultWriterEnsureReadyPromiseRejected(*writer*, *error*) nothrow §

1. If *writer*.[[readyPromise]].[[PromiseState]] is `"pending"`, reject *writer*.[[readyPromise]] with *error*.
2. Otherwise, set *writer*.[[readyPromise]] to a promise rejected with *error*.
3. Set *writer*.[[readyPromise]].[[PromiseIsHandled]] to **true**.

4.6.7. WritableStreamDefaultWriterGetDesiredSize (*writer*) nothrow §

1. Let *stream* be *writer*.[[ownerWritableStream]].
2. Let *state* be *stream*.[[state]].
3. If *state* is `"errored"` or `"erroring"`, return **null**.
4. If *state* is `"closed"`, return **0**.
5. Return ! WritableStreamDefaultControllerGetDesiredSize(*stream*.[[writableStreamController]]).

4.6.8. WritableStreamDefaultWriterRelease (*writer*) nothrow §

1. Let *stream* be *writer*.[[ownerWritableStream]].
2. Assert: *stream* is not **undefined**.
3. Assert: *stream*.[[writer]] is *writer*.
4. Let *releasedError* be a new **TypeError**.
5. Perform ! WritableStreamDefaultWriterEnsureReadyPromiseRejected(*writer*, *releasedError*).
6. Perform ! WritableStreamDefaultWriterEnsureClosedPromiseRejected(*writer*, *releasedError*).
7. Set *stream*.[[writer]] to **undefined**.
8. Set *writer*.[[ownerWritableStream]] to **undefined**.

4.6.9. WritableStreamDefaultWriterWrite (*writer*, *chunk*) nothrow §

1. Let *stream* be *writer*.[[ownerWritableStream]].
2. Assert: *stream* is not **undefined**.
3. Let *controller* be *stream*.[[writableStreamController]].
4. Let *chunkSize* be ! WritableStreamDefaultControllerGetChunkSize(*controller*, *chunk*).
5. If *stream* is not equal to *writer*.[[ownerWritableStream]], return a promise rejected with a **TypeError** exception.
6. Let *state* be *stream*.[[state]].
7. If *state* is `"errored"`, return a promise rejected with *stream*.[[storedError]].
8. If ! WritableStreamCloseQueuedOrInFlight(*stream*) is **true** or *state* is `"closed"`, return a promise rejected with a **TypeError** exception indicating that the stream is closing or closed.
9. If *state* is `"erroring"`, return a promise rejected with *stream*.[[storedError]].
10. Assert: *state* is `"writable"`.
11. Let promise be ! WritableStreamAddWriteRequest(*stream*).

File an issue about the selected text

12. Perform ! WritableStreamDefaultControllerWrite(*controller*, *chunk*, *chunkSize*).
13. Return *promise*.

4.7. Class `WritableStreamDefaultController`

The `WritableStreamDefaultController` class has methods that allow control of a `WritableStream`'s state. When constructing a `WritableStream`, the underlying sink is given a corresponding `WritableStreamDefaultController` instance to manipulate.

4.7.1. Class definition §

This section is non-normative.

If one were to write the `WritableStreamDefaultController` class in something close to the syntax of [ECMASCRIPT], it would look like

```
class WritableStreamDefaultController {
  constructor() // always throws

  error(e)
}
```

4.7.2. Internal slots §

Instances of `WritableStreamDefaultController` are created with the internal slots described in the following table:

Internal Slot	Description (*non-normative*)
[[abortAlgorithm]]	A promise-returning algorithm, taking one argument (the abort reason), which communicates a requested abort to the underlying sink
[[closeAlgorithm]]	A promise-returning algorithm which communicates a requested close to the underlying sink
[[controlledWritableStream]]	The `WritableStream` instance controlled
[[queue]]	A List representing the stream's internal queue of chunks
[[queueTotalSize]]	The total size of all the chunks stored in [[queue]] (see §6.2 Queue-with-sizes operations)
[[started]]	A boolean flag indicating whether the underlying sink has finished starting
[[strategyHWM]]	A number supplied by the creator of the stream as part of the stream's queuing strategy, indicating the point at which the stream will apply backpressure to its underlying sink
[[strategySizeAlgorithm]]	An algorithm to calculate the size of enqueued chunks, as part of the stream's queuing strategy
[[writeAlgorithm]]	A promise-returning algorithm, taking one argument (the chunk to write), which writes data to the underlying sink

4.7.3. new WritableStreamDefaultController()

Note

The `WritableStreamDefaultController` constructor cannot be used directly; `WritableStreamDefaultController` instances are created automatically during `WritableStream` construction.

1. Throw a **TypeError** exception.

4.7.4. Properties of the `WritableStreamDefaultController` prototype §

4.7.4.1. error(*e*)

Note

The `error` method will error the writable stream, making all future interactions with it fail with the given error `e`.

This method is rarely used, since usually it suffices to return a rejected promise from one of the underlying sink's methods. However, it can be useful for suddenly shutting down a stream in response to an event outside the normal lifecycle of interactions with the underlying sink.

1. If ! IsWritableStreamDefaultController(**this**) is **false**, throw a **TypeError** exception.
2. Let *state* be **this**.[[controlledWritableStream]].[[state]].

File an issue about the selected text e", return.

4. Perform ! WritableStreamDefaultControllerError(**this**, *e*).

4.7.5. Writable stream default controller internal methods §

The following are additional internal methods implemented by each `WritableStreamDefaultController` instance. The writable stream implementation will call into these.

Note

The reason these are in method form, instead of as abstract operations, is to make it clear that the writable stream implementation is decoupled from the controller implementation, and could in the future be expanded with other controllers, as long as those controllers implemented such internal methods. A similar scenario is seen for readable streams, where there actually are multiple controller types and as such the counterpart internal methods are used polymorphically.

4.7.5.1. [[AbortSteps]](*reason*) §

1. Let *result* be the result of performing **this**.[[abortAlgorithm]], passing *reason*.
2. Perform ! WritableStreamDefaultControllerClearAlgorithms(**this**).
3. Return *result*.

4.7.5.2. [[ErrorSteps]]() §

1. Perform ! ResetQueue(**this**).

4.8. Writable stream default controller abstract operations §

4.8.1. IsWritableStreamDefaultController (*x*) nothrow §

1. If Type(*x*) is not Object, return **false**.
2. If *x* does not have an [[controlledWritableStream]] internal slot, return **false**.
3. Return **true**.

4.8.2. SetUpWritableStreamDefaultController (*stream*, *controller*, *startAlgorithm*, *writeAlgorithm*, *closeAlgorithm*, *abortAlgorithm*, *highWaterMark*, *sizeAlgorithm*) throws §

1. Assert: ! IsWritableStream(*stream*) is **true**.
2. Assert: *stream*.[[writableStreamController]] is **undefined**.
3. Set *controller*.[[controlledWritableStream]] to *stream*.
4. Set *stream*.[[writableStreamController]] to *controller*.
5. Perform ! ResetQueue(*controller*).
6. Set *controller*.[[started]] to **false**.
7. Set *controller*.[[strategySizeAlgorithm]] to *sizeAlgorithm*.
8. Set *controller*.[[strategyHWM]] to *highWaterMark*.
9. Set *controller*.[[writeAlgorithm]] to *writeAlgorithm*.
10. Set *controller*.[[closeAlgorithm]] to *closeAlgorithm*.
11. Set *controller*.[[abortAlgorithm]] to *abortAlgorithm*.
12. Let *backpressure* be ! WritableStreamDefaultControllerGetBackpressure(*controller*).
13. Perform ! WritableStreamUpdateBackpressure(*stream*, *backpressure*).
14. Let *startResult* be the result of performing *startAlgorithm*. (This may throw an exception.)
15. Let *startPromise* be a promise resolved with *startResult*.
16. Upon fulfillment of *startPromise*,
    a. Assert: *stream*.[[state]] is `"writable"` or `"erroring"`.
    b. Set *controller*.[[started]] to **true**.
    c. Perform ! WritableStreamDefaultControllerAdvanceQueueIfNeeded(*controller*).
17. Upon rejection of *startPromise* with reason *r*,
    a. Assert: *stream*.[[state]] is `"writable"` or `"erroring"`.
    b. Set *controller*.[[started]] to **true**.
    c. Perform ! WritableStreamDealWithRejection(*stream*, *r*).

File an issue about the selected text

4.8.3. SetUpWritableStreamDefaultControllerFromUnderlyingSink (*stream*, *underlyingSink*, *highWaterMark*, *sizeAlgorithm*) throws §

1. Assert: *underlyingSink* is not **undefined**.
2. Let *controller* be ObjectCreate(the original value of `WritableStreamDefaultController`'s `prototype` property).
3. Let *startAlgorithm* be the following steps:
    a. Return ? InvokeOrNoop(*underlyingSink*, `"start"`, « *controller* »).
4. Let *writeAlgorithm* be ? CreateAlgorithmFromUnderlyingMethod(*underlyingSink*, `"write"`, **1**, « *controller* »).
5. Let *closeAlgorithm* be ? CreateAlgorithmFromUnderlyingMethod(*underlyingSink*, `"close"`, **0**, « »).
6. Let *abortAlgorithm* be ? CreateAlgorithmFromUnderlyingMethod(*underlyingSink*, `"abort"`, **1**, « »).
7. Perform ? SetUpWritableStreamDefaultController(*stream*, *controller*, *startAlgorithm*, *writeAlgorithm*, *closeAlgorithm*, *abortAlgorithm*, *highWaterMark*, *sizeAlgorithm*).

4.8.4. WritableStreamDefaultControllerClearAlgorithms (*controller*) nothrow §

This abstract operation is called once the stream is closed or errored and the algorithms will not be executed any more. By removing the algorithm references it permits the underlying sink object to be garbage collected even if the `WritableStream` itself is still referenced.

Note

The results of this algorithm are not currently observable, but could become so if JavaScript eventually adds weak references. But even without that factor, implementations will likely want to include similar steps.

Note

This operation will be performed multiple times in some edge cases. After the first time it will do nothing.

1. Set *controller*.[[writeAlgorithm]] to **undefined**.
2. Set *controller*.[[closeAlgorithm]] to **undefined**.
3. Set *controller*.[[abortAlgorithm]] to **undefined**.
4. Set *controller*.[[strategySizeAlgorithm]] to **undefined**.

4.8.5. WritableStreamDefaultControllerClose (*controller*) nothrow §

1. Perform ! EnqueueValueWithSize(*controller*, `"close"`, **0**).
2. Perform ! WritableStreamDefaultControllerAdvanceQueueIfNeeded(*controller*).

4.8.6. WritableStreamDefaultControllerGetChunkSize (*controller*, *chunk*) nothrow §

1. Let *returnValue* be the result of performing *controller*.[[strategySizeAlgorithm]], passing in *chunk*, and interpreting the result as an ECMAScript completion value.
2. If *returnValue* is an abrupt completion,
    a. Perform ! WritableStreamDefaultControllerErrorIfNeeded(*controller*, *returnValue*.[[Value]]).
    b. Return 1.
3. Return *returnValue*.[[Value]].

4.8.7. WritableStreamDefaultControllerGetDesiredSize (*controller*) nothrow §

1. Return *controller*.[[strategyHWM]] − *controller*.[[queueTotalSize]].

4.8.8. WritableStreamDefaultControllerWrite (*controller*, *chunk*, *chunkSize*) nothrow §

1. Let *writeRecord* be Record {[[chunk]]: *chunk*}.
2. Let *enqueueResult* be EnqueueValueWithSize(*controller*, *writeRecord*, *chunkSize*).
3. If *enqueueResult* is an abrupt completion,
    a. Perform ! WritableStreamDefaultControllerErrorIfNeeded(*controller*, *enqueueResult*.[[Value]]).
    b. Return.
4. Let *stream* be *controller*.[[controlledWritableStream]].
5. If ! WritableStreamCloseQueuedOrInFlight(*stream*) is **false** and *stream*.[[state]] is `"writable"`,
    e be ! WritableStreamDefaultControllerGetBackpressure(*controller*).

    b. Perform ! WritableStreamUpdateBackpressure(*stream*, *backpressure*).
6. Perform ! WritableStreamDefaultControllerAdvanceQueueIfNeeded(*controller*).

4.8.9. WritableStreamDefaultControllerAdvanceQueueIfNeeded (*controller*) nothrow §

1. Let *stream* be *controller*.[[controlledWritableStream]].
2. If *controller*.[[started]] is **false**, return.
3. If *stream*.[[inFlightWriteRequest]] is not **undefined**, return.
4. Let *state* be *stream*.[[state]].
5. Assert: *state* is not `"closed"` or `"errored"`.
6. If *state* is `"erroring"`,
    a. Perform ! WritableStreamFinishErroring(*stream*).
    b. Return.
7. If *controller*.[[queue]] is empty, return.
8. Let *writeRecord* be ! PeekQueueValue(*controller*).
9. If *writeRecord* is `"close"`, perform ! WritableStreamDefaultControllerProcessClose(*controller*).
10. Otherwise, perform ! WritableStreamDefaultControllerProcessWrite(*controller*, *writeRecord*.[[chunk]]).

4.8.10. WritableStreamDefaultControllerErrorIfNeeded (*controller*, *error*) nothrow §

1. If *controller*.[[controlledWritableStream]].[[state]] is `"writable"`, perform ! WritableStreamDefaultControllerError(*controller*, *error*).

4.8.11. WritableStreamDefaultControllerProcessClose (*controller*) nothrow §

1. Let *stream* be *controller*.[[controlledWritableStream]].
2. Perform ! WritableStreamMarkCloseRequestInFlight(*stream*).
3. Perform ! DequeueValue(*controller*).
4. Assert: *controller*.[[queue]] is empty.
5. Let *sinkClosePromise* be the result of performing *controller*.[[closeAlgorithm]].
6. Perform ! WritableStreamDefaultControllerClearAlgorithms(*controller*).
7. Upon fulfillment of *sinkClosePromise*,
    a. Perform ! WritableStreamFinishInFlightClose(*stream*).
8. Upon rejection of *sinkClosePromise* with reason *reason*,
    a. Perform ! WritableStreamFinishInFlightCloseWithError(*stream*, *reason*).

4.8.12. WritableStreamDefaultControllerProcessWrite (*controller*, *chunk*) nothrow §

1. Let *stream* be *controller*.[[controlledWritableStream]].
2. Perform ! WritableStreamMarkFirstWriteRequestInFlight(*stream*).
3. Let *sinkWritePromise* be the result of performing *controller*.[[writeAlgorithm]], passing in *chunk*.
4. Upon fulfillment of *sinkWritePromise*,
    a. Perform ! WritableStreamFinishInFlightWrite(*stream*).
    b. Let *state* be *stream*.[[state]].
    c. Assert: *state* is `"writable"` or `"erroring"`.
    d. Perform ! DequeueValue(*controller*).
    e. If ! WritableStreamCloseQueuedOrInFlight(*stream*) is **false** and *state* is `"writable"`,
        i. Let *backpressure* be ! WritableStreamDefaultControllerGetBackpressure(*controller*).
        ii. Perform ! WritableStreamUpdateBackpressure(*stream*, *backpressure*).
    f. Perform ! WritableStreamDefaultControllerAdvanceQueueIfNeeded(*controller*).
5. Upon rejection of *sinkWritePromise* with *reason*,
    a. If *stream*.[[state]] is `"writable"`, perform ! WritableStreamDefaultControllerClearAlgorithms(*controller*).
    b. Perform ! WritableStreamFinishInFlightWriteWithError(*stream*, *reason*).

4.8.13. WritableStreamDefaultControllerGetBackpressure (*controller*) nothrow §

1. Let *desiredSize* be ! WritableStreamDefaultControllerGetDesiredSize(*controller*).
2. Return *desiredSize* ≤ **0**.

File an issue about the selected text

4.8.14. WritableStreamDefaultControllerError (*controller*, *error*) nothrow §

1. Let *stream* be *controller*.[[controlledWritableStream]].
2. Assert: *stream*.[[state]] is `"writable"`.
3. Perform ! WritableStreamDefaultControllerClearAlgorithms(*controller*).
4. Perform ! WritableStreamStartErroring(*stream*, *error*).

5. Transform streams §

5.1. Using transform streams §

Example

The natural way to use a transform stream is to place it in a pipe between a readable stream and a writable stream. Chunks that travel from the readable stream to the writable stream will be transformed as they pass through the transform stream. Backpressure is respected, so data will not be read faster than it can be transformed and consumed.

```
readableStream
  .pipeThrough(transformStream)
  .pipeTo(writableStream)
  .then(() => console.log("All data successfully transformed!"))
  .catch(e => console.error("Something went wrong!", e));
```

Example

You can also use the `readable` and `writable` properties of a transform stream directly to access the usual interfaces of a readable stream and writable stream. In this example we supply data to the writable side of the stream using its writer interface. The readable side is then piped to `anotherWritableStream`.

```
const writer = transformStream.writable.getWriter();
writer.write("input chunk");
transformStream.readable.pipeTo(anotherWritableStream);
```

Example

One use of identity transform streams is to easily convert between readable and writable streams. For example, the `fetch()` API accepts a readable stream request body, but it can be more convenient to write data for uploading via a writable stream interface. Using an identity transform stream addresses this:

```
const { writable, readable } = new TransformStream();
fetch("...", { body: readable }).then(response => /* ... */);

const writer = writable.getWriter();
writer.write(new Uint8Array([0x73, 0x74, 0x72, 0x65, 0x61, 0x6D, 0x73, 0x21]));
writer.close();
```

Another use of identity transform streams is to add additional buffering to a pipe. In this example we add extra buffering between `readableStream` and `writableStream`.

```
const writableStrategy = new ByteLengthQueuingStrategy({ highWaterMark: 1024 * 1024 });

readableStream
  .pipeThrough(new TransformStream(undefined, writableStrategy))
  .pipeTo(writableStream);
```

5.2. Class `TransformStream`

5.2.1. Class definition §

This section is non-normative.

If one were to write the `TransformStream` class in something close to the syntax of [ECMASCRIPT], it would look like

```
class TransformStream {
  constructor(transformer = {}, writableStrategy = {}, readableStrategy = {})

  get readable()
```

File an issue about the selected text

```
    get writable()
  }
```

5.2.2. Internal slots §

Instances of `TransformStream` are created with the internal slots described in the following table:

Internal Slot	Description (*non-normative*)
[[backpressure]]	Whether there was backpressure on [[readable]] the last time it was observed
[[backpressureChangePromise]]	A promise which is fulfilled and replaced every time the value of [[backpressure]] changes
[[readable]]	The `ReadableStream` instance controlled by this object
[[transformStreamController]]	A `TransformStreamDefaultController` created with the ability to control [[readable]] and [[writable]]; also used for the IsTransformStream brand check
[[writable]]	The `WritableStream` instance controlled by this object

5.2.3. new TransformStream(*transformer* = {}, *writableStrategy* = {}, *readableStrategy* = {})

Note

The `transformer` argument represents the transformer, as described in §5.2.4 Transformer API.

The `writableStrategy` and `readableStrategy` arguments are the queuing strategy objects for the writable and readable sides respectively. These are used in the construction of the `WritableStream` and `ReadableStream` objects and can be used to add buffering to a `TransformStream`, in order to smooth out variations in the speed of the transformation, or to increase the amount of buffering in a pipe. If they are not provided, the default behavior will be the same as a `CountQueuingStrategy`, with respective high water marks of 1 and 0.

1. Let *writableSizeFunction* be ? GetV(*writableStrategy*, `"size"`).
2. Let *writableHighWaterMark* be ? GetV(*writableStrategy*, `"highWaterMark"`).
3. Let *readableSizeFunction* be ? GetV(*readableStrategy*, `"size"`).
4. Let *readableHighWaterMark* be ? GetV(*readableStrategy*, `"highWaterMark"`).
5. Let *writableType* be ? GetV(*transformer*, `"writableType"`).
6. If *writableType* is not **undefined**, throw a **RangeError** exception.
7. Let *writableSizeAlgorithm* be ? MakeSizeAlgorithmFromSizeFunction(*writableSizeFunction*).
8. If *writableHighWaterMark* is **undefined**, set *writableHighWaterMark* to **1**.
9. Set *writableHighWaterMark* to ? ValidateAndNormalizeHighWaterMark(*writableHighWaterMark*).
10. Let *readableType* be ? GetV(*transformer*, `"readableType"`).
11. If *readableType* is not **undefined**, throw a **RangeError** exception.
12. Let *readableSizeAlgorithm* be ? MakeSizeAlgorithmFromSizeFunction(*readableSizeFunction*).
13. If *readableHighWaterMark* is **undefined**, set *readableHighWaterMark* to **0**.
14. Set *readableHighWaterMark* be ? ValidateAndNormalizeHighWaterMark(*readableHighWaterMark*).
15. Let *startPromise* be a new promise.
16. Perform ! InitializeTransformStream(**this**, *startPromise*, *writableHighWaterMark*, *writableSizeAlgorithm*, *readableHighWaterMark*, *readableSizeAlgorithm*).
17. Perform ? SetUpTransformStreamDefaultControllerFromTransformer(**this**, *transformer*).
18. Let *startResult* be ? InvokeOrNoop(*transformer*, `"start"`, « **this**.[[transformStreamController]] »).
19. Resolve *startPromise* with *startResult*.

5.2.4. Transformer API §

This section is non-normative.

The `TransformStream()` constructor accepts as its first argument a JavaScript object representing the transformer. Such objects can contain any of the following methods:

`start(controller)`

A function that is called immediately during creation of the `TransformStream`.

Typically this is used to enqueue prefix chunks, using `controller.enqueue()`. Those chunks will be read from the readable side but don't depend on any writes to the writable side.

If this initial process is asynchronous, for example because it takes some effort to acquire the prefix chunks, the function can return a promise to signal success or failure; a rejected promise will error the stream. Any thrown exceptions will be re-thrown by the `TransformStream()`

transform(chunk, controller)

A function called when a new chunk originally written to the writable side is ready to be transformed. The stream implementation guarantees that this function will be called only after previous transforms have succeeded, and never before `start()` has completed or after `flush()` has been called.

This function performs the actual transformation work of the transform stream. It can enqueue the results using `controller.enqueue()`. This permits a single chunk written to the writable side to result in zero or multiple chunks on the readable side, depending on how many times `controller.enqueue()` is called. §8.9 A transform stream that replaces template tags demonstrates this by sometimes enqueuing zero chunks.

If the process of transforming is asynchronous, this function can return a promise to signal success or failure of the transformation. A rejected promise will error both the readable and writable sides of the transform stream.

If no `transform()` is supplied, the identity transform is used, which enqueues chunks unchanged from the writable side to the readable side.

flush(controller)

A function called after all chunks written to the writable side have been transformed by successfully passing through `transform()`, and the writable side is about to be closed.

Typically this is used to enqueue suffix chunks to the readable side, before that too becomes closed. An example can be seen in §8.9 A transform stream that replaces template tags.

If the flushing process is asynchronous, the function can return a promise to signal success or failure; the result will be communicated to the caller of `stream.writable.write()`. Additionally, a rejected promise will error both the readable and writable sides of the stream. Throwing an exception is treated the same as returning a rejected promise.

(Note that there is no need to call `controller.terminate()` inside `flush()`; the stream is already in the process of successfully closing down, and terminating it would be counterproductive.)

The `controller` object passed to `start()`, `transform()`, and `flush()` is an instance of `TransformStreamDefaultController`, and has the ability to enqueue chunks to the readable side, or to terminate or error the stream.

5.2.5. Properties of the `TransformStream` prototype §

5.2.5.1. get readable

Note

The `readable` getter gives access to the readable side of the transform stream.

1. If ! IsTransformStream(**this**) is **false**, throw a **TypeError** exception.
2. Return **this**.[[readable]].

5.2.5.2. get writable

Note

The `writable` getter gives access to the writable side of the transform stream.

1. If ! IsTransformStream(**this**) is **false**, throw a **TypeError** exception.
2. Return **this**.[[writable]].

5.3. General transform stream abstract operations §

5.3.1. CreateTransformStream (*startAlgorithm*, *transformAlgorithm*, *flushAlgorithm* [, *writableHighWaterMark* [, *writableSizeAlgorithm* [, *readableHighWaterMark* [, *readableSizeAlgorithm*]]]]) throws §

This abstract operation is meant to be called from other specifications that wish to create `TransformStream` instances. The *transformAlgorithm* and *flushAlgorithm* algorithms must return promises; if supplied, *writableHighWaterMark* and *readableHighWaterMark* must be non-negative, non-NaN numbers; and if supplied, *writableSizeAlgorithm* and *readableSizeAlgorithm* must be algorithms accepting chunk objects and returning numbers.

Note

CreateTransformStream throws an exception if and only if the supplied startAlgorithm *throws.*

1. If *writableHighWaterMark* was not passed, set it to **1**.
2. If *writableSizeAlgorithm* was not passed, set it to an algorithm that returns **1**.
3. If *readableHighWaterMark* was not passed, set it to **0**.

4. If *readableSizeAlgorithm* was not passed, set it to an algorithm that returns **1**.
5. Assert: ! IsNonNegativeNumber(*writableHighWaterMark*) is **true**.
6. Assert: ! IsNonNegativeNumber(*readableHighWaterMark*) is **true**.
7. Let *stream* be ObjectCreate(the original value of `TransformStream`'s `prototype` property).
8. Let *startPromise* be a new promise.
9. Perform ! InitializeTransformStream(*stream*, *startPromise*, *writableHighWaterMark*, *writableSizeAlgorithm*, *readableHighWaterMark*, *readableSizeAlgorithm*).
10. Let *controller* be ObjectCreate(the original value of `TransformStreamDefaultController`'s `prototype` property).
11. Perform ! SetUpTransformStreamDefaultController(*stream*, *controller*, *transformAlgorithm*, *flushAlgorithm*).
12. Let *startResult* be the result of performing *startAlgorithm*. (This may throw an exception.)
13. Resolve *startPromise* with *startResult*.
14. Return *stream*.

5.3.2. InitializeTransformStream (*stream*, *startPromise*, *writableHighWaterMark*, *writableSizeAlgorithm*, *readableHighWaterMark*, *readableSizeAlgorithm*) nothrow §

1. Let *startAlgorithm* be an algorithm that returns *startPromise*.
2. Let *writeAlgorithm* be the following steps, taking a *chunk* argument:
    a. Return ! TransformStreamDefaultSinkWriteAlgorithm(*stream*, *chunk*).
3. Let *abortAlgorithm* be the following steps, taking a *reason* argument:
    a. Return ! TransformStreamDefaultSinkAbortAlgorithm(*stream*, *reason*).
4. Let *closeAlgorithm* be the following steps:
    a. Return ! TransformStreamDefaultSinkCloseAlgorithm(*stream*).
5. Set *stream*.[[writable]] to ! CreateWritableStream(*startAlgorithm*, *writeAlgorithm*, *closeAlgorithm*, *abortAlgorithm*, *writableHighWaterMark*, *writableSizeAlgorithm*).
6. Let *pullAlgorithm* be the following steps:
    a. Return ! TransformStreamDefaultSourcePullAlgorithm(*stream*).
7. Let *cancelAlgorithm* be the following steps, taking a *reason* argument:
    a. Perform ! TransformStreamErrorWritableAndUnblockWrite(*stream*, *reason*).
    b. Return a promise resolved with **undefined**.
8. Set *stream*.[[readable]] to ! CreateReadableStream(*startAlgorithm*, *pullAlgorithm*, *cancelAlgorithm*, *readableHighWaterMark*, *readableSizeAlgorithm*).
9. Set *stream*.[[backpressure]] and *stream*.[[backpressureChangePromise]] to **undefined**.

Note

*The [[backpressure]] slot is set to **undefined** so that it can be initialized by TransformStreamSetBackpressure. Alternatively, implementations can use a strictly boolean value for [[backpressure]] and change the way it is initialized. This will not be visible to user code so long as the initialization is correctly completed before* transformer's `start()` *method is called.*

10. Perform ! TransformStreamSetBackpressure(*stream*, **true**).
11. Set *stream*.[[transformStreamController]] to **undefined**.

5.3.3. IsTransformStream (*x*) nothrow §

1. If Type(*x*) is not Object, return **false**.
2. If *x* does not have a [[transformStreamController]] internal slot, return **false**.
3. Return **true**.

5.3.4. TransformStreamError (*stream*, *e*) nothrow §

1. Perform ! ReadableStreamDefaultControllerError(*stream*.[[readable]].[[readableStreamController]], *e*).
2. Perform ! TransformStreamErrorWritableAndUnblockWrite(*stream*, *e*).

Note

This operation works correctly when one or both sides are already errored. As a result, calling algorithms do not need to check stream states when responding to an error condition.

5.3.5. TransformStreamErrorWritableAndUnblockWrite (*stream*, *e*) nothrow §

1. Perform ! TransformStreamDefaultControllerClearAlgorithms(*stream*.[[transformStreamController]]).

2. Perform ! WritableStreamDefaultControllerErrorIfNeeded(*stream*.[[writable]].[[writableStreamController]], *e*).
3. If *stream*.[[backpressure]] is **true**, perform ! TransformStreamSetBackpressure(*stream*, **false**).

Note

The TransformStreamDefaultSinkWriteAlgorithm abstract operation could be waiting for the promise stored in the [[backpressureChangePromise]] slot to resolve. This call to TransformStreamSetBackpressure ensures that the promise always resolves.

5.3.6. TransformStreamSetBackpressure (*stream*, *backpressure*) nothrow §

1. Assert: *stream*.[[backpressure]] is not *backpressure*.
2. If *stream*.[[backpressureChangePromise]] is not **undefined**, resolve stream.[[backpressureChangePromise]] with **undefined**.
3. Set *stream*.[[backpressureChangePromise]] to a new promise.
4. Set *stream*.[[backpressure]] to *backpressure*.

5.4. Class `TransformStreamDefaultController`

The `TransformStreamDefaultController` class has methods that allow manipulation of the associated `ReadableStream` and `WritableStream`. When constructing a `TransformStream`, the transformer object is given a corresponding `TransformStreamDefaultController` instance to manipulate.

5.4.1. Class definition §

This section is non-normative.

If one were to write the `TransformStreamDefaultController` class in something close to the syntax of [ECMASCRIPT], it would look like

```
class TransformStreamDefaultController {
  constructor() // always throws

  get desiredSize()

  enqueue(chunk)
  error(reason)
  terminate()
}
```

5.4.2. Internal slots §

Instances of `TransformStreamDefaultController` are created with the internal slots described in the following table:

Internal Slot	Description (*non-normative*)
[[controlledTransformStream]]	The `TransformStream` instance controlled; also used for the IsTransformStreamDefaultController brand check
[[flushAlgorithm]]	A promise-returning algorithm which communicates a requested close to the transformer
[[transformAlgorithm]]	A promise-returning algorithm, taking one argument (the chunk to transform), which requests the transformer perform its transformation

5.4.3. new TransformStreamDefaultController()

Note

The `TransformStreamDefaultController` constructor cannot be used directly; `TransformStreamDefaultController` instances are created automatically during `TransformStream` construction.

1. Throw a **TypeError** exception.

5.4.4. Properties of the `TransformStreamDefaultController` prototype §

5.4.4.1. get desiredSize

Note

The `desiredSize` getter returns the desired size to fill the readable side's internal queue. It can be negative, if the queue is over-full.

1. If ! IsTransformStreamDefaultController(**this**) is **false**, throw a **TypeError** exception.
2. Let *readableController* be **this**.[[controlledTransformStream]].[[readable]].[[readableStreamController]].
3. Return ! ReadableStreamDefaultControllerGetDesiredSize(*readableController*).

5.4.4.2. enqueue(*chunk*)

Note

The `enqueue` method will enqueue a given chunk in the readable side.

1. If ! IsTransformStreamDefaultController(**this**) is **false**, throw a **TypeError** exception.
2. Perform ? TransformStreamDefaultControllerEnqueue(**this**, *chunk*).

5.4.4.3. error(*reason*)

Note

The `error` method will error both the readable side and the writable side of the controlled transform stream, making all future interactions fail with the given `reason`. Any chunks queued for transformation will be discarded.

1. If ! IsTransformStreamDefaultController(**this**) is **false**, throw a **TypeError** exception.
2. Perform ! TransformStreamDefaultControllerError(**this**, *reason*).

5.4.4.4. terminate()

Note

The `terminate` method will close the readable side and error the writable side of the controlled transform stream. This is useful when the transformer only needs to consume a portion of the chunks written to the writable side.

1. If ! IsTransformStreamDefaultController(**this**) is **false**, throw a **TypeError** exception.
2. Perform ! TransformStreamDefaultControllerTerminate(**this**).

5.5. Transform stream default controller abstract operations §

5.5.1. IsTransformStreamDefaultController (*x*) nothrow §

1. If Type(*x*) is not Object, return **false**.
2. If *x* does not have an [[controlledTransformStream]] internal slot, return **false**.
3. Return **true**.

5.5.2. SetUpTransformStreamDefaultController (*stream*, *controller*, *transformAlgorithm*, *flushAlgorithm*) nothrow §

1. Assert: ! IsTransformStream(*stream*) is **true**.
2. Assert: *stream*.[[transformStreamController]] is **undefined**.
3. Set *controller*.[[controlledTransformStream]] to *stream*.
4. Set *stream*.[[transformStreamController]] to *controller*.
5. Set *controller*.[[transformAlgorithm]] to *transformAlgorithm*.
6. Set *controller*.[[flushAlgorithm]] to *flushAlgorithm*.

5.5.3. SetUpTransformStreamDefaultControllerFromTransformer (*stream*, *transformer*) throws §

1. Assert: *transformer* is not **undefined**.
2. Let *controller* be ObjectCreate(the original value of `TransformStreamDefaultController`'s `prototype` property).

File an issue about the selected text ıe the following steps, taking a *chunk* argument:

a. Let *result* be TransformStreamDefaultControllerEnqueue(*controller*, *chunk*).
b. If *result* is an abrupt completion, return a promise rejected with *result*.[[Value]].
c. Otherwise, return a promise resolved with **undefined**.

4. Let *transformMethod* be ? GetV(*transformer*, `"transform"`).
5. If *transformMethod* is not **undefined**,
   a. If ! IsCallable(*transformMethod*) is **false**, throw a **TypeError** exception.
   b. Set *transformAlgorithm* to the following steps, taking a *chunk* argument:
      i. Return ! PromiseCall(*transformMethod*, *transformer*, « *chunk*, *controller* »).
6. Let *flushAlgorithm* be ? CreateAlgorithmFromUnderlyingMethod(*transformer*, `"flush"`, **0**, « controller »).
7. Perform ! SetUpTransformStreamDefaultController(*stream*, *controller*, *transformAlgorithm*, *flushAlgorithm*).

5.5.4. TransformStreamDefaultControllerClearAlgorithms (*controller*) nothrow §

This abstract operation is called once the stream is closed or errored and the algorithms will not be executed any more. By removing the algorithm references it permits the transformer object to be garbage collected even if the `TransformStream` itself is still referenced.

Note

The results of this algorithm are not currently observable, but could become so if JavaScript eventually adds weak references. But even without that factor, implementations will likely want to include similar steps.

1. Set *controller*.[[transformAlgorithm]] to **undefined**.
2. Set *controller*.[[flushAlgorithm]] to **undefined**.

5.5.5. TransformStreamDefaultControllerEnqueue (*controller*, *chunk*) throws §

This abstract operation can be called by other specifications that wish to enqueue chunks in the readable side, in the same way a developer would enqueue chunks using the stream's associated controller object. Specifications should *not* do this to streams they did not create.

1. Let *stream* be *controller*.[[controlledTransformStream]].
2. Let *readableController* be *stream*.[[readable]].[[readableStreamController]].
3. If ! ReadableStreamDefaultControllerCanCloseOrEnqueue(*readableController*) is **false**, throw a **TypeError** exception.
4. Let *enqueueResult* be ReadableStreamDefaultControllerEnqueue(*readableController*, *chunk*).
5. If *enqueueResult* is an abrupt completion,
   a. Perform ! TransformStreamErrorWritableAndUnblockWrite(*stream*, *enqueueResult*.[[Value]]).
   b. Throw *stream*.[[readable]].[[storedError]].
6. Let *backpressure* be ! ReadableStreamDefaultControllerHasBackpressure(*readableController*).
7. If *backpressure* is not *stream*.[[backpressure]],
   a. Assert: *backpressure* is **true**.
   b. Perform ! TransformStreamSetBackpressure(*stream*, **true**).

5.5.6. TransformStreamDefaultControllerError (*controller*, *e*) nothrow §

This abstract operation can be called by other specifications that wish to move a transform stream to an errored state, in the same way a developer would error a stream using its associated controller object. Specifications should *not* do this to streams they did not create.

1. Perform ! TransformStreamError(*controller*.[[controlledTransformStream]], *e*).

5.5.7. TransformStreamDefaultControllerPerformTransform (*controller*, *chunk*) nothrow §

1. Let *transformPromise* be the result of performing *controller*.[[transformAlgorithm]], passing *chunk*.
2. Return the result of transforming *transformPromise* with a rejection handler that, when called with argument *r*, performs the following steps:
   a. Perform ! TransformStreamError(*controller*.[[controlledTransformStream]], *r*).
   b. Throw *r*.

5.5.8. TransformStreamDefaultControllerTerminate (*controller*) nothrow §

This abstract operation can be called by other specifications that wish to terminate a transform stream, in the same way a developer-created stream controller object. Specifications should *not* do this to streams they did not create.

1. Let *stream* be *controller*.[[controlledTransformStream]].
2. Let *readableController* be *stream*.[[readable]].[[readableStreamController]].
3. If ! ReadableStreamDefaultControllerCanCloseOrEnqueue(*readableController*) is **true**, perform ! ReadableStreamDefaultControllerClose(*readableController*).
4. Let *error* be a **TypeError** exception indicating that the stream has been terminated.
5. Perform ! TransformStreamErrorWritableAndUnblockWrite(*stream*, *error*).

5.6. Transform stream default sink abstract operations §

5.6.1. TransformStreamDefaultSinkWriteAlgorithm (*stream*, *chunk*) nothrow §

1. Assert: *stream*.[[writable]].[[state]] is `"writable"`.
2. Let *controller* be *stream*.[[transformStreamController]].
3. If *stream*.[[backpressure]] is **true**,
    a. Let *backpressureChangePromise* be *stream*.[[backpressureChangePromise]].
    b. Assert: *backpressureChangePromise* is not **undefined**.
    c. Return the result of transforming *backpressureChangePromise* with a fulfillment handler which performs the following steps:
        i. Let *writable* be *stream*.[[writable]].
        ii. Let *state* be *writable*.[[state]].
        iii. If *state* is `"erroring"`, throw *writable*.[[storedError]].
        iv. Assert: *state* is `"writable"`.
        v. Return ! TransformStreamDefaultControllerPerformTransform(*controller*, *chunk*).
4. Return ! TransformStreamDefaultControllerPerformTransform(*controller*, *chunk*).

5.6.2. TransformStreamDefaultSinkAbortAlgorithm (*stream*, *reason*) nothrow §

1. Perform ! TransformStreamError(*stream*, *reason*).
2. Return a promise resolved with **undefined**.

5.6.3. TransformStreamDefaultSinkCloseAlgorithm(*stream*) nothrow §

1. Let *readable* be *stream*.[[readable]].
2. Let *controller* be *stream*.[[transformStreamController]].
3. Let *flushPromise* be the result of performing *controller*.[[flushAlgorithm]].
4. Perform ! TransformStreamDefaultControllerClearAlgorithms(*controller*).
5. Return the result of transforming *flushPromise* with:
    a. A fulfillment handler that performs the following steps:
        i. If *readable*.[[state]] is `"errored"`, throw *readable*.[[storedError]].
        ii. Let *readableController* be *readable*.[[readableStreamController]].
        iii. If ! ReadableStreamDefaultControllerCanCloseOrEnqueue(*readableController*) is **true**, perform ! ReadableStreamDefaultControllerClose(*readableController*).
    b. A rejection handler that, when called with argument *r*, performs the following steps:
        i. Perform ! TransformStreamError(*stream*, *r*).
        ii. Throw *readable*.[[storedError]].

5.7. Transform stream default source abstract operations §

5.7.1. TransformStreamDefaultSourcePullAlgorithm(*stream*) nothrow §

1. Assert: *stream*.[[backpressure]] is **true**.
2. Assert: *stream*.[[backpressureChangePromise]] is not **undefined**.
3. Perform ! TransformStreamSetBackpressure(*stream*, **false**).
4. Return *stream*.[[backpressureChangePromise]].

6. Other stream APIs and operations §

6.1. Queuing strategies §

6.1.1. The queuing strategy API §

This section is non-normative.

The `ReadableStream()`, `WritableStream()`, and `TransformStream()` constructors all accept at least one argument representing an appropriate queuing strategy for the stream being created. Such objects contain the following properties:

***`size(chunk)`* (non-byte streams only)**

A function that computes and returns the finite non-negative size of the given chunk value.

The result is used to determine backpressure, manifesting via the appropriate `desiredSize` property: either `defaultController.desiredSize`, `byteController.desiredSize`, or `writer.desiredSize`, depending on where the queuing strategy is being used. For readable streams, it also governs when the underlying source's `pull()` method is called.

This function has to be idempotent and not cause side effects; very strange results can occur otherwise.

For readable byte streams, this function is not used, as chunks are always measured in bytes.

`highWaterMark`

A non-negative number indicating the high water mark of the stream using this queuing strategy.

Any object with these properties can be used when a queuing strategy object is expected. However, we provide two built-in queuing strategy classes that provide a common vocabulary for certain cases: `ByteLengthQueuingStrategy` and `CountQueuingStrategy`.

6.1.2. Class `ByteLengthQueuingStrategy`

A common queuing strategy when dealing with bytes is to wait until the accumulated `byteLength` properties of the incoming chunks reaches a specified high-water mark. As such, this is provided as a built-in queuing strategy that can be used when constructing streams.

Example

When creating a readable stream or writable stream, you can supply a byte-length queuing strategy directly:

```
const stream = new ReadableStream(
  { ... },
  new ByteLengthQueuingStrategy({ highWaterMark: 16 * 1024 })
);
```

In this case, 16 KiB worth of chunks can be enqueued by the readable stream's underlying source before the readable stream implementation starts sending backpressure signals to the underlying source.

```
const stream = new WritableStream(
  { ... },
  new ByteLengthQueuingStrategy({ highWaterMark: 32 * 1024 })
);
```

In this case, 32 KiB worth of chunks can be accumulated in the writable stream's internal queue, waiting for previous writes to the underlying sink to finish, before the writable stream starts sending backpressure signals to any producers.

Note

It is not necessary to use `ByteLengthQueuingStrategy` with readable byte streams, as they always measure chunks in bytes. Attempting to construct a byte stream with a `ByteLengthQueuingStrategy` will fail.

6.1.2.1. Class definition §

File an issue about the selected text

This section is non-normative.

If one were to write the ByteLengthQueuingStrategy class in something close to the syntax of [ECMASCRIPT], it would look like

```
class ByteLengthQueuingStrategy {
  constructor({ highWaterMark })
  size(chunk)
}
```

Each ByteLengthQueuingStrategy instance will additionally have an own data property `highWaterMark` set by its constructor.

6.1.2.2. new ByteLengthQueuingStrategy({ *highWaterMark* })

Note

The constructor takes a non-negative number for the high-water mark, and stores it on the object as a property.

1. Perform ! CreateDataProperty(**this**, `"highWaterMark"`, *highWaterMark*).

6.1.2.3. Properties of the ByteLengthQueuingStrategy prototype §

6.1.2.3.1. size(chunk)

Note

*The `size` method returns the given chunk's `byteLength` property. (If the chunk doesn't have one, it will return **undefined**, causing the stream using this strategy to error.)*

*This method is intentionally generic; it does not require that its **this** value be a `ByteLengthQueuingStrategy` object.*

1. Return ? GetV(*chunk*, `"byteLength"`).

6.1.3. Class `CountQueuingStrategy`

A common queuing strategy when dealing with streams of generic objects is to simply count the number of chunks that have been accumulated so far, waiting until this number reaches a specified high-water mark. As such, this strategy is also provided out of the box.

Example

When creating a readable stream or writable stream, you can supply a count queuing strategy directly:

```
const stream = new ReadableStream(
  { ... },
  new CountQueuingStrategy({ highWaterMark: 10 })
);
```

In this case, 10 chunks (of any kind) can be enqueued by the readable stream's underlying source before the readable stream implementation starts sending backpressure signals to the underlying source.

```
const stream = new WritableStream(
  { ... },
  new CountQueuingStrategy({ highWaterMark: 5 })
);
```

In this case, five chunks (of any kind) can be accumulated in the writable stream's internal queue, waiting for previous writes to the underlying sink to finish, before the writable stream starts sending backpressure signals to any producers.

6.1.3.1. Class definition §

This section is non-normative.

If one were to write the CountQueuingStrategy class in something close to the syntax of [ECMASCRIPT], it would look like

```
class CountQueuingStrategy {
  constructor({ highWaterMark })
  size(chunk)
}
```

Each CountQueuingStrategy instance will additionally have an own data property highWaterMark set by its constructor.

6.1.3.2. new CountQueuingStrategy({ *highWaterMark* })

Note

The constructor takes a non-negative number for the high-water mark, and stores it on the object as a property.

1. Perform ! CreateDataProperty(**this**, "highWaterMark", *highWaterMark*).

6.1.3.3. Properties of the CountQueuingStrategy prototype §

6.1.3.3.1. size()

Note

The size method returns one always, so that the total queue size is a count of the number of chunks in the queue.

*This method is intentionally generic; it does not require that its **this** value be a CountQueuingStrategy object.*

1. Return **1**.

6.2. Queue-with-sizes operations §

The streams in this specification use a "queue-with-sizes" data structure to store queued up values, along with their determined sizes. Various specification objects contain a queue-with-sizes, represented by the object having two paired internal slots, always named [[queue]] and [[queueTotalSize]]. [[queue]] is a List of Records with [[value]] and [[size]] fields, and [[queueTotalSize]] is a JavaScript Number, i.e. a double-precision floating point number.

The following abstract operations are used when operating on objects that contain queues-with-sizes, in order to ensure that the two internal slots stay synchronized.

⚠Warning!

Due to the limited precision of floating-point arithmetic, the framework specified here, of keeping a running total in the [[queueTotalSize]] slot, is not equivalent to adding up the size of all chunks in [[queue]]. (However, this only makes a difference when there is a huge ($\sim 10^{15}$) variance in size between chunks, or when trillions of chunks are enqueued.)

6.2.1. DequeueValue (*container*) nothrow §

1. Assert: *container* has [[queue]] and [[queueTotalSize]] internal slots.
2. Assert: *container*.[[queue]] is not empty.
3. Let *pair* be the first element of *container*.[[queue]].
4. Remove *pair* from *container*.[[queue]], shifting all other elements downward (so that the second becomes the first, and so on).
5. Set *container*.[[queueTotalSize]] to *container*.[[queueTotalSize]] − *pair*.[[size]].
6. If *container*.[[queueTotalSize]] < **0**, set *container*.[[queueTotalSize]] to **0**. (This can occur due to rounding errors.)
7. Return *pair*.[[value]].

6.2.2. EnqueueValueWithSize (*container*, *value*, *size*) throws §

1. Assert: *container* has [[queue]] and [[queueTotalSize]] internal slots.
2. Let *size* be ? ToNumber(*size*).
3. If ! IsFiniteNonNegativeNumber(*size*) is **false**, throw a **RangeError** exception.
4. Append Record {[[value]]: *value*, [[size]]: *size*} as the last element of *container*.[[queue]].
5. Set *container*.[[queueTotalSize]] to *container*.[[queueTotalSize]] + *size*.

File an issue about the selected text

6.2.3. PeekQueueValue (*container*) nothrow §

1. Assert: *container* has [[queue]] and [[queueTotalSize]] internal slots.
2. Assert: *container*.[[queue]] is not empty.
3. Let *pair* be the first element of *container*.[[queue]].
4. Return *pair*.[[value]].

6.2.4. ResetQueue (*container*) nothrow §

1. Assert: *container* has [[queue]] and [[queueTotalSize]] internal slots.
2. Set *container*.[[queue]] to a new empty List.
3. Set *container*.[[queueTotalSize]] to **0**.

6.3. Miscellaneous operations §

A few abstract operations are used in this specification for utility purposes. We define them here.

6.3.1. CreateAlgorithmFromUnderlyingMethod (*underlyingObject*, *methodName*, *algoArgCount*, *extraArgs*) throws §

1. Assert: *underlyingObject* is not **undefined**.
2. Assert: ! IsPropertyKey(*methodName*) is **true**.
3. Assert: *algoArgCount* is **0** or **1**.
4. Assert: *extraArgs* is a List.
5. Let *method* be ? GetV(*underlyingObject*, *methodName*).
6. If *method* is not **undefined**,
    a. If ! IsCallable(*method*) is **false**, throw a **TypeError** exception.
    b. If *algoArgCount* is **0**, return an algorithm that performs the following steps:
        i. Return ! PromiseCall(*method*, *underlyingObject*, *extraArgs*).
    c. Otherwise, return an algorithm that performs the following steps, taking an *arg* argument:
        i. Let *fullArgs* be a List consisting of *arg* followed by the elements of *extraArgs* in order.
        ii. Return ! PromiseCall(*method*, *underlyingObject*, *fullArgs*).
7. Return an algorithm which returns a promise resolved with **undefined**.

6.3.2. InvokeOrNoop (*O*, *P*, *args*) throws §

Note

*InvokeOrNoop is a slight modification of the [ECMASCRIPT] Invoke abstract operation to return **undefined** when the method is not present.*

1. Assert: *O* is not **undefined**.
2. Assert: ! IsPropertyKey(*P*) is **true**.
3. Assert: *args* is a List.
4. Let *method* be ? GetV(*O*, *P*).
5. If *method* is **undefined**, return **undefined**.
6. Return ? Call(*method*, *O*, *args*).

6.3.3. IsFiniteNonNegativeNumber (*v*) nothrow §

1. If ! IsNonNegativeNumber(*v*) is **false**, return **false**.
2. If *v* is **+∞**, return **false**.
3. Return **true**.

6.3.4. IsNonNegativeNumber (*v*) nothrow §

1. If Type(*v*) is not Number, return **false**.
2. If *v* is **NaN**, return **false**.

3. If *v* < **0**, return **false**.
4. Return **true**.

6.3.5. PromiseCall (*F*, *V*, *args*) nothrow §

Note

PromiseCall is a variant of promise-calling that works on methods.

1. Assert: ! IsCallable(*F*) is **true**.
2. Assert: *V* is not **undefined**.
3. Assert: *args* is a List.
4. Let *returnValue* be Call(*F*, *V*, *args*).
5. If *returnValue* is an abrupt completion, return a promise rejected with *returnValue*.[[Value]].
6. Otherwise, return a promise resolved with *returnValue*.[[Value]].

6.3.6. TransferArrayBuffer (*O*) nothrow §

1. Assert: Type(*O*) is Object.
2. Assert: *O* has an [[ArrayBufferData]] internal slot.
3. Assert: ! IsDetachedBuffer(*O*) is **false**.
4. Let *arrayBufferData* be *O*.[[ArrayBufferData]].
5. Let *arrayBufferByteLength* be *O*.[[ArrayBufferByteLength]].
6. Perform ! DetachArrayBuffer(*O*).
7. Return a new `ArrayBuffer` object (created in the current Realm Record) whose [[ArrayBufferData]] internal slot value is *arrayBufferData* and whose [[ArrayBufferByteLength]] internal slot value is *arrayBufferByteLength*.

6.3.7. ValidateAndNormalizeHighWaterMark (*highWaterMark*) throws §

1. Set *highWaterMark* to ? ToNumber(*highWaterMark*).
2. If *highWaterMark* is **NaN** or *highWaterMark* < **0**, throw a **RangeError** exception.

   Note

   +∞ is explicitly allowed as a valid high water mark. It causes backpressure to never be applied.

3. Return *highWaterMark*.

6.3.8. MakeSizeAlgorithmFromSizeFunction (*size*) throws §

1. If *size* is **undefined**, return an algorithm that returns **1**.
2. If ! IsCallable(*size*) is **false**, throw a **TypeError** exception.
3. Return an algorithm that performs the following steps, taking a *chunk* argument:
   a. Return ? Call(*size*, **undefined**, « *chunk* »).

File an issue about the selected text

7. Global properties §

The following constructors must be exposed on the global object as data properties of the same name:

- `ReadableStream`
- `WritableStream`
- `TransformStream`
- `ByteLengthQueuingStrategy`
- `CountQueuingStrategy`

The attributes of these properties must be { [[Writable]]: **true**, [[Enumerable]]: **false**, [[Configurable]]: **true** }.

Note

The `ReadableStreamDefaultReader`, `ReadableStreamBYOBReader`, `ReadableStreamDefaultController`, `ReadableByteStreamController`, `WritableStreamDefaultWriter`, `WritableStreamDefaultController`, and `TransformStreamDefaultController` classes are specifically not exposed, as they are not independently useful.

8. Examples of creating streams §

This section, and all its subsections, are non-normative.

The previous examples throughout the standard have focused on how to use streams. Here we show how to create a stream, using the `ReadableStream` or `WritableStream` constructors.

8.1. A readable stream with an underlying push source (no backpressure support) §

The following function creates readable streams that wrap `WebSocket` instances [HTML], which are push sources that do not support backpressure signals. It illustrates how, when adapting a push source, usually most of the work happens in the `start()` function.

```
function makeReadableWebSocketStream(url, protocols) {
  const ws = new WebSocket(url, protocols);
  ws.binaryType = "arraybuffer";

  return new ReadableStream({
    start(controller) {
      ws.onmessage = event => controller.enqueue(event.data);
      ws.onclose = () => controller.close();
      ws.onerror = () => controller.error(new Error("The WebSocket errored!"));
    },

    cancel() {
      ws.close();
    }
  });
}
```

We can then use this function to create readable streams for a web socket, and pipe that stream to an arbitrary writable stream:

```
const webSocketStream = makeReadableWebSocketStream("wss://example.com:443/", "protocol");

webSocketStream.pipeTo(writableStream)
  .then(() => console.log("All data successfully written!"))
  .catch(e => console.error("Something went wrong!", e));
```

Note

This specific style of wrapping a web socket interprets web socket messages directly as chunks. This can be a convenient abstraction, for example when piping to a writable stream or transform stream for which each web socket message makes sense as a chunk to consume or transform.

However, often when people talk about "adding streams support to web sockets", they are hoping instead for a new capability to send an individual web socket message in a streaming fashion, so that e.g. a file could be transferred in a single message without holding all of its contents in memory on the client side. To accomplish this goal, we'd instead want to allow individual web socket messages to themselves be `ReadableStream` instances. That isn't what we show in the above example.

For more background, see this discussion.

8.2. A readable stream with an underlying push source and backpressure support §

The following function returns readable streams that wrap "backpressure sockets," which are hypothetical objects that have the same API as web sockets, but also provide the ability to pause and resume the flow of data with their `readStop` and `readStart` methods. In doing so, this example shows how to apply backpressure to underlying sources that support it.

```
function makeReadableBackpressureSocketStream(host, port) {
  const socket = createBackpressureSocket(host, port);
```

```
    return new ReadableStream({
      start(controller) {
        socket.ondata = event => {
          controller.enqueue(event.data);

          if (controller.desiredSize <= 0) {
            // The internal queue is full, so propagate
            // the backpressure signal to the underlying source.
            socket.readStop();
          }
        };

        socket.onend = () => controller.close();
        socket.onerror = () => controller.error(new Error("The socket errored!"));
      },

      pull() {
        // This is called if the internal queue has been emptied, but the
        // stream's consumer still wants more data. In that case, restart
        // the flow of data if we have previously paused it.
        socket.readStart();
      },

      cancel() {
        socket.close();
      }
    });
  }
```

We can then use this function to create readable streams for such "backpressure sockets" in the same way we do for web sockets. This time, however, when we pipe to a destination that cannot accept data as fast as the socket is producing it, or if we leave the stream alone without reading from it for some time, a backpressure signal will be sent to the socket.

8.3. A readable byte stream with an underlying push source (no backpressure support) §

The following function returns readable byte streams that wraps a hypothetical UDP socket API, including a promise-returning `select2()` method that is meant to be evocative of the POSIX select(2) system call.

Since the UDP protocol does not have any built-in backpressure support, the backpressure signal given by `desiredSize` is ignored, and the stream ensures that when data is available from the socket but not yet requested by the developer, it is enqueued in the stream's internal queue, to avoid overflow of the kernel-space queue and a consequent loss of data.

This has some interesting consequences for how consumers interact with the stream. If the consumer does not read data as fast as the socket produces it, the chunks will remain in the stream's internal queue indefinitely. In this case, using a BYOB reader will cause an extra copy, to move the data from the stream's internal queue to the developer-supplied buffer. However, if the consumer consumes the data quickly enough, a BYOB reader will allow zero-copy reading directly into developer-supplied buffers.

(You can imagine a more complex version of this example which uses `desiredSize` to inform an out-of-band backpressure signaling mechanism, for example by sending a message down the socket to adjust the rate of data being sent. That is left as an exercise for the reader.)

```
const DEFAULT_CHUNK_SIZE = 65536;

function makeUDPSocketStream(host, port) {
  const socket = createUDPSocket(host, port);

  return new ReadableStream({
    type: "bytes",

    start(controller) {
      readRepeatedly().catch(e => controller.error(e));

      function readRepeatedly() {
        return socket.select2().then(() => {
          // Since the socket can become readable even when there's
          // no pending BYOB requests, we need to handle both cases.
          Read;
```

```
            if (controller.byobRequest) {
              const v = controller.byobRequest.view;
              bytesRead = socket.readInto(v.buffer, v.byteOffset, v.byteLength);
              controller.byobRequest.respond(bytesRead);
            } else {
              const buffer = new ArrayBuffer(DEFAULT_CHUNK_SIZE);
              bytesRead = socket.readInto(buffer, 0, DEFAULT_CHUNK_SIZE);
              controller.enqueue(new Uint8Array(buffer, 0, bytesRead));
            }

            if (bytesRead === 0) {
              controller.close();
              return;
            }

            return readRepeatedly();
          });
        }
      },

      cancel() {
        socket.close();
      }
    });
  }
```

`ReadableStream` instances returned from this function can now vend BYOB readers, with all of the aforementioned benefits and caveats.

8.4. A readable stream with an underlying pull source §

The following function returns readable streams that wrap portions of the Node.js file system API (which themselves map fairly directly to C's `fopen`, `fread`, and `fclose` trio). Files are a typical example of pull sources. Note how in contrast to the examples with push sources, most of the work here happens on-demand in the `pull()` function, and not at startup time in the `start()` function.

```
  const fs = require("pr/fs"); // https://github.com/jden/pr
  const CHUNK_SIZE = 1024;

  function makeReadableFileStream(filename) {
    let fd;
    let position = 0;

    return new ReadableStream({
      start() {
        return fs.open(filename, "r").then(result => {
          fd = result;
        });
      },

      pull(controller) {
        const buffer = new ArrayBuffer(CHUNK_SIZE);

        return fs.read(fd, buffer, 0, CHUNK_SIZE, position).then(bytesRead => {
          if (bytesRead === 0) {
            return fs.close(fd).then(() => controller.close());
          } else {
            position += bytesRead;
            controller.enqueue(new Uint8Array(buffer, 0, bytesRead));
          }
        });
      },

      cancel() {
        return fs.close(fd);
      }
    });
  }
```

File an issue about the selected text

We can then create and use readable streams for files just as we could before for sockets.

8.5. A readable byte stream with an underlying pull source §

The following function returns readable byte streams that allow efficient zero-copy reading of files, again using the Node.js file system API. Instead of using a predetermined chunk size of 1024, it attempts to fill the developer-supplied buffer, allowing full control.

```
const fs = require("pr/fs"); // https://github.com/jden/pr
const DEFAULT_CHUNK_SIZE = 1024;

function makeReadableByteFileStream(filename) {
  let fd;
  let position = 0;

  return new ReadableStream({
    type: "bytes",

    start() {
      return fs.open(filename, "r").then(result => {
        fd = result;
      });
    },

    pull(controller) {
      // Even when the consumer is using the default reader, the auto-allocation
      // feature allocates a buffer and passes it to us via byobRequest.
      const v = controller.byobRequest.view;

      return fs.read(fd, v.buffer, v.byteOffset, v.byteLength, position).then(bytesRead => {
        if (bytesRead === 0) {
          return fs.close(fd).then(() => controller.close());
        } else {
          position += bytesRead;
          controller.byobRequest.respond(bytesRead);
        }
      });
    },

    cancel() {
      return fs.close(fd);
    },

    autoAllocateChunkSize: DEFAULT_CHUNK_SIZE
  });
}
```

With this in hand, we can create and use BYOB readers for the returned `ReadableStream`. But we can also create default readers, using them in the same simple and generic manner as usual. The adaptation between the low-level byte tracking of the underlying byte source shown here, and the higher-level chunk-based consumption of a default reader, is all taken care of automatically by the streams implementation. The auto-allocation feature, via the `autoAllocateChunkSize` option, even allows us to write less code, compared to the manual branching in §8.3 A readable byte stream with an underlying push source (no backpressure support).

8.6. A writable stream with no backpressure or success signals §

The following function returns a writable stream that wraps a `WebSocket` [HTML]. Web sockets do not provide any way to tell when a given chunk of data has been successfully sent (without awkward polling of `bufferedAmount`, which we leave as an exercise to the reader). As such, this writable stream has no ability to communicate accurate backpressure signals or write success/failure to its producers. That is, the promises returned by its writer's `write()` method and `ready` getter will always fulfill immediately.

```
function makeWritableWebSocketStream(url, protocols) {
  const ws = new WebSocket(url, protocols);

  return new WritableStream({
```

```
    start(controller) {
      ws.onerror = () => {
        controller.error(new Error("The WebSocket errored!"));
        ws.onclose = null;
      };
      ws.onclose = () => controller.error(new Error("The server closed the connection
unexpectedly!"));
      return new Promise(resolve => ws.onopen = resolve);
    },

    write(chunk) {
      ws.send(chunk);
      // Return immediately, since the web socket gives us no easy way to tell
      // when the write completes.
    },

    close() {
      return closeWS(1000);
    },

    abort(reason) {
      return closeWS(4000, reason && reason.message);
    },
  });

  function closeWS(code, reasonString) {
    return new Promise((resolve, reject) => {
      ws.onclose = e => {
        if (e.wasClean) {
          resolve();
        } else {
          reject(new Error("The connection was not closed cleanly"));
        }
      };
      ws.close(code, reasonString);
    });
  }
}
```

We can then use this function to create writable streams for a web socket, and pipe an arbitrary readable stream to it:

```
const webSocketStream = makeWritableWebSocketStream("wss://example.com:443/", "protocol");

readableStream.pipeTo(webSocketStream)
  .then(() => console.log("All data successfully written!"))
  .catch(e => console.error("Something went wrong!", e));
```

Note

See the earlier note about this style of wrapping web sockets into streams.

8.7. A writable stream with backpressure and success signals §

The following function returns writable streams that wrap portions of the Node.js file system API (which themselves map fairly directly to C's `fopen`, `fwrite`, and `fclose` trio). Since the API we are wrapping provides a way to tell when a given write succeeds, this stream will be able to communicate backpressure signals as well as whether an individual write succeeded or failed.

```
const fs = require("pr/fs"); // https://github.com/jden/pr

function makeWritableFileStream(filename) {
  let fd;

  return new WritableStream({
    start() {
      return fs.open(filename, "w").then(result => {
        fd = result;
```

File an issue about the selected text

```
    },

    write(chunk) {
      return fs.write(fd, chunk, 0, chunk.length);
    },

    close() {
      return fs.close(fd);
    },

    abort() {
      return fs.close(fd);
    }
  });
}
```

We can then use this function to create a writable stream for a file, and write individual chunks of data to it:

```
const fileStream = makeWritableFileStream("/example/path/on/fs.txt");
const writer = fileStream.getWriter();

writer.write("To stream, or not to stream\n");
writer.write("That is the question\n");

writer.close()
  .then(() => console.log("chunks written and stream closed successfully!"))
  .catch(e => console.error(e));
```

Note that if a particular call to `fs.write` takes a longer time, the returned promise will fulfill later. In the meantime, additional writes can be queued up, which are stored in the stream's internal queue. The accumulation of chunks in this queue can change the stream to return a pending promise from the `ready` getter, which is a signal to producers that they would benefit from backing off and stopping writing, if possible.

The way in which the writable stream queues up writes is especially important in this case, since as stated in the documentation for `fs.write`, "it is unsafe to use `fs.write` multiple times on the same file without waiting for the [promise]." But we don't have to worry about that when writing the `makeWritableFileStream` function, since the stream implementation guarantees that the underlying sink's `write()` method will not be called until any promises returned by previous calls have fulfilled!

8.8. A { readable, writable } stream pair wrapping the same underlying resource §

The following function returns an object of the form `{ readable, writable }`, with the `readable` property containing a readable stream and the `writable` property containing a writable stream, where both streams wrap the same underlying web socket resource. In essence, this combines §8.1 A readable stream with an underlying push source (no backpressure support) and §8.6 A writable stream with no backpressure or success signals.

While doing so, it illustrates how you can use JavaScript classes to create reusable underlying sink and underlying source abstractions.

```
function streamifyWebSocket(url, protocol) {
  const ws = new WebSocket(url, protocols);
  ws.binaryType = "arraybuffer";

  return {
    readable: new ReadableStream(new WebSocketSource(ws)),
    writable: new WritableStream(new WebSocketSink(ws))
  };
}

class WebSocketSource {
  constructor(ws) {
    this._ws = ws;
  }

  start(controller) {
    this._ws.onmessage = event => controller.enqueue(event.data);
    this._ws.onclose = () => controller.close();

    this._ws.addEventListener("error", () => {
      controller.error(new Error("The WebSocket errored!"));
```

```
      });
    }

    cancel() {
      this._ws.close();
    }
  }

  class WebSocketSink {
    constructor(ws) {
      this._ws = ws;
    }

    start(controller) {
      this._ws.onclose = () => controller.error(new Error("The server closed the connection
unexpectedly!"));
      this._ws.addEventListener("error", () => {
        controller.error(new Error("The WebSocket errored!"));
        this._ws.onclose = null;
      });

      return new Promise(resolve => this._ws.onopen = resolve);
    }

    write(chunk) {
      this._ws.send(chunk);
    }

    close() {
      return this._closeWS(1000);
    }

    abort(reason) {
      return this._closeWS(4000, reason && reason.message);
    }

    _closeWS(code, reasonString) {
      return new Promise((resolve, reject) => {
        this._ws.onclose = e => {
          if (e.wasClean) {
            resolve();
          } else {
            reject(new Error("The connection was not closed cleanly"));
          }
        };
        this._ws.close(code, reasonString);
      });
    }
  }
```

We can then use the objects created by this function to communicate with a remote web socket, using the standard stream APIs:

```
const streamyWS = streamifyWebSocket("wss://example.com:443/", "protocol");
const writer = streamyWS.writable.getWriter();
const reader = streamyWS.readable.getReader();

writer.write("Hello");
writer.write("web socket!");

reader.read().then(({ value, done }) => {
  console.log("The web socket says: ", value);
});
```

Note how in this setup canceling the `readable` side will implicitly close the `writable` side, and similarly, closing or aborting the `writable` side will implicitly close the `readable` side.

Note

See the earlier note about this style of wrapping web sockets into streams.

File an issue about the selected text

8.9. A transform stream that replaces template tags §

It's often useful to substitute tags with variables on a stream of data, where the parts that need to be replaced are small compared to the overall data size. This example presents a simple way to do that. It maps strings to strings, transforming a template like `"Time: {{time}} Message: {{message}}"` to `"Time: 15:36 Message: hello"` assuming that `{ time: "15:36", message: "hello" }` was passed in the `substitutions` parameter to `LipFuzzTransformer`.

This example also demonstrates one way to deal with a situation where a chunk contains partial data that cannot be transformed until more data is received. In this case, a partial template tag will be accumulated in the `partialChunk` instance variable until either the end of the tag is found or the end of the stream is reached.

```
class LipFuzzTransformer {
  constructor(substitutions) {
    this.substitutions = substitutions;
    this.partialChunk = "";
    this.lastIndex = undefined;
  }

  transform(chunk, controller) {
    chunk = this.partialChunk + chunk;
    this.partialChunk = "";
    // lastIndex is the index of the first character after the last substitution.
    this.lastIndex = 0;
    chunk = chunk.replace(/\{\{([a-zA-Z0-9_-]+)\}\}/g, this.replaceTag.bind(this));
    // Regular expression for an incomplete template at the end of a string.
    const partialAtEndRegexp = /\{(\{([a-zA-Z0-9_-]+(\})?)?)?$/g;
    // Avoid looking at any characters that have already been substituted.
    partialAtEndRegexp.lastIndex = this.lastIndex;
    this.lastIndex = undefined;
    const match = partialAtEndRegexp.exec(chunk);
    if (match) {
      this.partialChunk = chunk.substring(match.index);
      chunk = chunk.substring(0, match.index);
    }
    controller.enqueue(chunk);
  }

  flush(controller) {
    if (this.partialChunk.length > 0) {
      controller.enqueue(this.partialChunk);
    }
  }

  replaceTag(match, p1, offset) {
    let replacement = this.substitutions[p1];
    if (replacement === undefined) {
      replacement = "";
    }
    this.lastIndex = offset + replacement.length;
    return replacement;
  }
}
```

In this case we define the transformer to be passed to the `TransformStream` constructor as a class. This is useful when there is instance data to track.

The class would be used in code like:

```
const data = { userName, displayName, icon, date };
const ts = new TransformStream(new LipFuzzTransformer(data));

fetchEvent.respondWith(
  fetch(fetchEvent.request.url).then(response => {
    const transformedBody = response.body
      // Decode the binary-encoded response to string
      .pipeThrough(new TextDecoderStream())
      // Apply the LipFuzzTransformer
      .pipeThrough(ts)
      // Encode the transformed string
      .pipeThrough(new TextEncoderStream());
```

```
        return new Response(transformedBody);
      })
    );
```

⚠Warning!

For simplicity, `LipFuzzTransformer` performs unescaped text substitutions. In real applications, a template system that performs context-aware escaping is good practice for security and robustness.

8.10. A transform stream created from a sync mapper function §

The following function allows creating new `TransformStream` instances from synchronous "mapper" functions, of the type you would normally pass to `Array.prototype.map`. It demonstrates that the API is concise even for trivial transforms.

```
function mapperTransformStream(mapperFunction) {
  return new TransformStream({
    transform(chunk, controller) {
      controller.enqueue(mapperFunction(chunk));
    }
  });
}
```

This function can then be used to create a `TransformStream` that uppercases all its inputs:

```
const ts = mapperTransformStream(chunk => chunk.toUpperCase());
const writer = ts.writable.getWriter();
const reader = ts.readable.getReader();

writer.write("No need to shout");

// Logs "NO NEED TO SHOUT":
reader.read().then(({ value }) => console.log(value));
```

Although a synchronous transform never causes backpressure itself, it will only transform chunks as long as there is no backpressure, so resources will not be wasted.

Exceptions error the stream in a natural way:

```
const ts = mapperTransformStream(chunk => JSON.parse(chunk));
const writer = ts.writable.getWriter();
const reader = ts.readable.getReader();

writer.write("[1, ");

// Logs a SyntaxError, twice:
reader.read().catch(e => console.error(e));
writer.write("{}").catch(e => console.error(e));
```

Conventions §

This specification depends on the Infra Standard. [INFRA]

This specification uses algorithm conventions very similar to those of [ECMASCRIPT], whose rules should be used to interpret it (apart from the exceptions enumerated below). In particular, the objects specified here should be treated as built-in objects. For example, their `name` and `length` properties are derived as described by that specification, as are the default property descriptor values and the treatment of missing, **undefined**, or surplus arguments.

We also depart from the [ECMASCRIPT] conventions in the following ways, mostly for brevity. It is hoped (and vaguely planned) that the conventions of ECMAScript itself will evolve in these ways.

- We prefix section headings with `new` to indicate they are defining constructors; when doing so, we assume that NewTarget will be checked before the algorithm starts.
- We use the default argument notation `= {}` in a couple of cases, meaning that before the algorithm starts, **undefined** (including the implicit **undefined** when no argument is provided) is instead treated as a new object created as if by ObjectCreate(%ObjectPrototype%). (This object may then be destructured, if combined with the below destructuring convention.)
- We use destructuring notation in function and method declarations, and assume that DestructuringAssignmentEvaluation was performed appropriately before the algorithm starts.
- We use "**this**" instead of "**this** value".
- We use the shorthand phrases from the [PROMISES-GUIDE] to operate on promises at a higher level than the ECMAScript spec does.

It's also worth noting that, as in [ECMASCRIPT], all numbers are represented as double-precision floating point values, and all arithmetic operations performed on them must be done in the standard way for such values.

Acknowledgments §

The editors would like to thank Anne van Kesteren, AnthumChris, Arthur Langereis, Ben Kelly, Bert Belder, Brian di Palma, Calvin Metcalf, Dominic Tarr, Ed Hager, Forbes Lindesay, Forrest Norvell, Gary Blackwood, Gorgi Kosev, Gus Caplan, 贺师俊 (hax), Isaac Schlueter, isonmad, Jake Archibald, Jake Verbaten, Janessa Det, Jason Orendorff, Jens Nockert, Lennart Grahl, Mangala Sadhu Sangeet Singh Khalsa, Marcos Caceres, Marvin Hagemeister, Mattias Buelens, Michael Mior, Mihai Potra, Romain Bellessort, Simon Menke, Stephen Sugden, Surma, Tab Atkins, Tanguy Krotoff, Thorsten Lorenz, Till Schneidereit, Tim Caswell, Trevor Norris, tzik, Will Chan, Youenn Fablet, 平野裕 (Yutaka Hirano), and Xabier Rodríguez for their contributions to this specification. Community involvement in this specification has been above and beyond; we couldn't have done it without you.

This standard is written by Adam Rice (Google, ricea@chromium.org), Domenic Denicola (Google, d@domenic.me), and 吉野剛史 (Takeshi Yoshino, Google, tyoshino@chromium.org).

Index §

Terms defined by this specification §

File an issue about the selected text

File an issue about the selected text

- underlying byte source, in §2.1
- underlying sink, in §2.2
- underlying source, in §2.1
- view, in §3.12.4
- writable, in §5.2.5.1
- writable side, in §2.3
- WritableStream, in §4.1
- writable stream, in §2.2
- WritableStreamDefaultController, in §4.6.9
- WritableStreamDefaultController(), in §4.7.2
- WritableStreamDefaultWriter, in §4.4.14
- WritableStreamDefaultWriter(stream), in §4.5.2
- WritableStream(underlyingSink, strategy), in §4.2.2
- writable stream writer, in §2.6
- write(chunk), in §4.5.4.6
- write(chunk, controller), in §4.2.4
- writer, in §2.6

Terms defined by reference §

- [DOM] defines the following terms:
  - AbortController
  - AbortSignal
  - aborted flag
  - add
  - remove
- [ECMASCRIPT] defines the following terms:
  - %AsyncIteratorPrototype%
  - %Uint8Array%
  - ArrayBuffer
  - AsyncIterator
  - CreateIterResultObject
  - DataView
  - DestructuringAssignmentEvaluation
  - Invoke
  - Number
  - TypeError
  - Uint8Array
  - map
  - the typed array constructors table
  - typed array
- [FETCH] defines the following terms:
  - Response
  - body
  - fetch(input)
- [HTML] defines the following terms:
  - StructuredDeserialize
  - StructuredSerialize
  - WebSocket
  - bufferedAmount
  - img
  - in parallel
  - serializable objects
  - transferable objects
  - unhandledrejection
- [INFRA] defines the following terms:
  - append
  - ordered set
- [PROMISES-GUIDE] defines the following terms:
  - a new promise
  - a promise rejected with
  - a promise resolved with
  - promise-calling
  - reject
  - resolve
  - transforming

    - upon rejection
    - waiting for all
- [SERVICE-WORKERS] defines the following terms:
    - fetch
- [WebIDL] defines the following terms:
    - AbortError
    - DOMException

References §

Normative References §

[DOM]

Anne van Kesteren. DOM Standard. Living Standard. URL: https://dom.spec.whatwg.org/

[ECMASCRIPT]

ECMAScript Language Specification. URL: https://tc39.github.io/ecma262/

[HTML]

Anne van Kesteren; et al. HTML Standard. Living Standard. URL: https://html.spec.whatwg.org/multipage/

[INFRA]

Anne van Kesteren; Domenic Denicola. Infra Standard. Living Standard. URL: https://infra.spec.whatwg.org/

[PROMISES-GUIDE]

Domenic Denicola. Writing Promise-Using Specifications. 16 February 2016. TAG Finding. URL: https://www.w3.org/2001/tag/doc/promises-guide

[WebIDL]

Boris Zbarsky. Web IDL. URL: https://heycam.github.io/webidl/

Informative References §

[FETCH]

Anne van Kesteren. Fetch Standard. Living Standard. URL: https://fetch.spec.whatwg.org/

[SERVICE-WORKERS]

Alex Russell; et al. Service Workers 1. URL: https://w3c.github.io/ServiceWorker/