Encoding

Living Standard — Last Updated 2 March 2021

Participate:

[GitHub whatwg/encoding](#) ([new issue](#), [open issues](#))
[IRC: #whatwg on Freenode](#)

Commits:

[GitHub whatwg/encoding/commits](#)
[Snapshot as of this commit](#)
[@encodings](#)

Tests:

[web-platform-tests encoding/](#) ([ongoing work](#))

Translations (non-normative):

[日本語](#)

Abstract

The Encoding Standard defines encodings and their JavaScript API.

Table of Contents

§ 1. Preface

The UTF-8 encoding is the most appropriate encoding for interchange of Unicode, the universal coded character set. Therefore for new protocols and formats, as well as existing formats deployed in new contexts, this specification requires (and defines) the UTF-8 encoding.

The other (legacy) encodings have been defined to some extent in the past. However, user agents have not always implemented them in the same way, have not always used the same labels, and often differ in dealing with undefined and former proprietary areas of encodings. This specification addresses those gaps so that new user agents do not have to reverse engineer encoding implementations and existing user agents can converge.

In particular, this specification defines all those encodings, their algorithms to go from bytes to scalar values and back, and their canonical names and identifying labels. This specification also defines an API to expose part of the encoding algorithms to JavaScript.

User agents have also significantly deviated from the labels listed in the IANA Character Sets registry. To stop spreading legacy encodings further, this specification is exhaustive about the aforementioned details and therefore has no need for the registry. In particular, this specification does not provide a mechanism for extending any aspect of encodings.

§ 2. Security background

There is a set of encoding security issues when the producer and consumer do not agree on the encoding in use, or on the way a given encoding is to be implemented. For instance, an attack was reported in 2011 where a Shift_JIS lead byte 0x82 was used to “mask” a 0x22 trail byte in a JSON resource of which an attacker could control some field. The producer did not see the problem even though this is an illegal byte combination. The consumer decoded it as a single U+FFFD and therefore changed the overall interpretation as U+0022 is an important delimiter. Decoders of encodings that use multiple bytes for scalar values now require that in case of an illegal byte combination, a scalar value in the range U+0000 to U+007F, inclusive, cannot be “masked”. For the aforementioned sequence the output would be U+FFFD U+0022. (As an unfortunate exception to this, the gb18030 decoder will “mask” up to one such byte at end-of-queue.)

This is a larger issue for encodings that map anything that is an ASCII byte to something that is not an ASCII code point, when there is no lead byte present. These are “ASCII-incompatible” encodings and other than ISO-2022-JP and UTF-16BE/LE, which are unfortunately required due to deployed content, they are not supported. (Investigation is ongoing whether more labels of other such encodings can be mapped to the replacement encoding, rather than the unknown encoding fallback.) An example attack is injecting carefully crafted content into a resource and then encouraging the user to override the encoding, resulting in, e.g., script execution.

Encoders used by URLs found in HTML and HTML’s form feature can also result in slight information loss when an encoding is used that cannot represent all scalar values. E.g., when a resource uses the windows-1252 encoding a server will not be able to distinguish between an end user entering “💩” and “💩” into a form.

The problems outlined here go away when exclusively using UTF-8, which is one of the many reasons that is now the mandatory encoding for all things.

Note

See also the Browser UI chapter.

§ 3. Terminology

This specification depends on the Infra Standard. [INFRA]

Hexadecimal numbers are prefixed with "0x".

In equations, all numbers are integers, addition is represented by "+", subtraction by "−", multiplication by "×", integer division by "/" (returns the quotient), modulo by "%" (returns the remainder of an integer division), logical left shifts by "<<", logical right shifts by ">>", bitwise AND by "&", and bitwise OR by "|".

For logical right shifts operands must have at least twenty-one bits precision.

An **I/O queue** is a type of list with items of a particular type (i.e., bytes or scalar values). **End-of-queue** is a special item that can be present in I/O queues of any type and it signifies that there are no more items in the queue.

Note

There are two ways to use an I/O queue: in immediate mode, to represent I/O data stored in memory, and in streaming mode, to represent data coming in from the network. Immediate queues have end-of-queue as their last item, whereas streaming queues need not have it, and so their read operation might block.

It is expected that streaming I/O queues will be created empty, and that new items will be pushed to it as data comes in from the network. When the underlying network stream closes, an end-of-queue item is to be pushed into the queue.

Since reading from a streaming I/O queue might block, streaming I/O queues are not to be used from an event loop. They are to be used in parallel instead.

To **read** an item from an I/O queue *ioQueue*, run these steps:

1. If *ioQueue* is empty, then wait until its size is at least 1.
2. If *ioQueue*[0] is end-of-queue, then return end-of-queue.
3. Remove *ioQueue*[0] and return it.

To read a number *number* of items from *ioQueue*, run these steps:

1. Let *readItems* be an empty list.
2. Perform the following step *number* times:
    1. Append to *readItems* the result of reading an item from *ioQueue*.
3. Remove end-of-queue from *readItems*.
4. Return *readItems*.

To **peek** a number *number* of items from an I/O queue *ioQueue*, run these steps:

1. Wait until either *ioQueue*'s size is equal to or greater than *number*, or *ioQueue* contains end-of-queue, whichever comes first.
2. Let *prefix* be an empty list.

3. For each *n* in the range 1 to *number*, inclusive:

    1. If *ioQueue*[*n*] is end-of-queue, break.

    2. Otherwise, append *ioQueue*[*n*] to *prefix*.

4. Return *prefix*.

To **push** an item *item* to an I/O queue *ioQueue*, run these steps:

1. If the last item in *ioQueue* is end-of-queue, then:

    1. If *item* is end-of-queue, do nothing.

    2. Otherwise, insert *item* before the last item in *ioQueue*.

2. Otherwise, append *item* to *ioQueue*.

To push a sequence of items to an I/O queue *ioQueue* is to push each item in the sequence to *ioQueue*, in the given order.

To **prepend** an item other than end-of-queue to an I/O queue, perform the normal list prepend operation. To prepend a sequence of items not containing end-of-queue, insert those items, in the given order, before the first item in the queue.

Example

Inserting the sequence of scalar value items `💩` in an I/O queue of scalar values "` hello world`", results in an I/O queue "`💩 hello world`". The next item to be read would be `&`.

To **convert** an I/O queue *ioQueue* into a list, string, or byte sequence, return the result of reading an indefinite number of items from *ioQueue*.

To **convert** a list, string, or byte sequence *input* into an I/O queue, run these steps:

1. Assert: if *input* is a list, then it does not contain end-of-queue.

2. Return an I/O queue containing the items in *input*, in order, followed by end-of-queue.

The Infra standard is expected to define some infrastructure around type conversions. See whatwg/infra issue #319. [INFRA]

Note

I/O queues are defined as lists, not queues, because they feature a prepend operation. However, this prepend operation is an internal detail of the algorithms in this specification, and is not to be used by other standards. Implementations are free to find alternative ways to implement such algorithms, as detailed in Implementation considerations.

§ 4. Encodings

An **encoding** defines a mapping from a scalar value sequence to a byte sequence (and vice versa). Each encoding has a **name**, and one or more **labels**.

Note

This specification defines three encodings with the same names as encoding schemes *defined in the Unicode standard: UTF-8, UTF-16LE, and UTF-16BE. The encodings differ from the* encoding schemes *by byte order mark (also known as BOM) handling not being part of the encodings themselves and instead being part of wrapper algorithms in this specification, whereas byte order mark handling is part of the definition of the* encoding schemes *in the Unicode Standard. UTF-8 used together with the UTF-8 decode algorithm matches the* encoding scheme *of the same name. This specification does not provide wrapper algorithms that would combine with UTF-16LE and UTF-16BE to match the similarly-named* encoding schemes*. [UNICODE]*

§ 4.1. Encoders and decoders

Each encoding has an associated **decoder** and most of them have an associated **encoder**. Instances of decoders and encoders have a **handler** algorithm and might also have state. A handler algorithm takes an input I/O queue and an item, and returns **finished**, one or more items, **error** optionally with a code point, or **continue**.

Note

The replacement and UTF-16BE/LE encodings have no encoder.

An **error mode** as used below is "`replacement`" or "`fatal`" for a decoder and "`fatal`" or "`html`" for an encoder.

Note

An XML processor would set error mode to "`fatal`"*. [XML]*

Note

"`html`" *exists as error mode due to HTML forms requiring a non-terminating legacy encoder. The* "`html`" *error mode causes a sequence to be emitted that cannot be distinguished from legitimate input and can therefore lead to silent data loss. Developers are strongly encouraged to use the UTF-8 encoding to prevent this from happening. [HTML]*

To **process a queue** given an encoding's decoder or encoder instance *encoderDecoder*, I/O queue *input*, I/O queue *output*, and error mode *mode*:

1. While true:
    1. Let *result* be the result of processing an item with the result of reading from *input*, *encoderDecoder*, *input*, *output*, and *mode*.
    2. If *result* is not continue, then return *result*.

To **process an item** given an item *item*, encoding's encoder or decoder instance *encoderDecoder*, I/O queue *input*, I/O queue *output*, and error mode *mode*:

1. Assert: if *encoderDecoder* is an encoder instance, *mode* is not "`replacement`".
2. Assert: if *encoderDecoder* is a decoder instance, *mode* is not "`html`".
3. Assert: if *encoderDecoder* is an encoder instance, *item* is not a surrogate.
4. Let *result* be the result of running *encoderDecoder*'s handler on *input* and *item*.
5. If *result* is finished:
    1. Push end-of-queue to *output*.
    2. Return *result*.
6. Otherwise, if *result* is one or more items:
    1. Assert: if *encoderDecoder* is a decoder instance, *result* does not contain any surrogates.
    2. Push *result* to *output*.
7. Otherwise, if *result* is an error, switch on *mode* and run the associated steps:

    ↪ **`"replacement"`**

    Push U+FFFD (�) to *output*.

    ↪ **`"html"`**

    Push 0x26 (&), 0x23 (#), followed by the shortest sequence of 0x30 (0) to 0x39 (9), inclusive, representing *result*'s code point's value in base ten, followed by 0x3B (;) to *output*.

    ↪ **`"fatal"`**

    Return *result*.
8. Return continue.

§ 4.2. Names and labels

The table below lists all encodings and their labels user agents must support. User agents must not support any other encodings or labels.

Note

For each encoding, ASCII-lowercasing its name yields one of its labels.

Authors must use the UTF-8 encoding and must use the ASCII case-insensitive "`utf-8`" label to identify it.

New protocols and formats, as well as existing formats deployed in new contexts, must use the UTF-8 encoding exclusively. If these protocols and formats need to expose the encoding's name or label, they must expose it as "`utf-8`".

To **get an encoding** from a string *label*, run these steps:

1. Remove any leading and trailing ASCII whitespace from *label*.
2. If *label* is an ASCII case-insensitive match for any of the labels listed in the table below, then return the corresponding encoding; otherwise return failure.

Note

This is a more basic and restrictive algorithm of mapping labels to encodings than section 1.4 of Unicode Technical Standard #22 prescribes, as that is necessary to be compatible with deployed content.

Name	Labels
The Encoding	
UTF-8	"unicode-1-1-utf-8"
	"unicode11utf8"
	"unicode20utf8"
	"utf-8"
	"utf8"
	"x-unicode20utf8"
Legacy single-byte encodings	
IBM866	"866"
	"cp866"
	"csibm866"
	"ibm866"
ISO-8859-2	"csisolatin2"
	"iso-8859-2"
	"iso-ir-101"
	"iso8859-2"
	"iso88592"
	"iso_8859-2"
	"iso_8859-2:1987"
	"l2"
	"latin2"
ISO-8859-3	"csisolatin3"
	"iso-8859-3"
	"iso-ir-109"
	"iso8859-3"
	"iso88593"
	"iso_8859-3"
	"iso_8859-3:1988"
	"l3"
	"latin3"
ISO-8859-4	"csisolatin4"
	"iso-8859-4"
	"iso-ir-110"
	"iso8859-4"
	"iso88594"
	"iso_8859-4"
	"iso_8859-4:1988"
	"l4"

Name	Labels
	"latin4"
ISO-8859-5	"csisolatincyrillic"
	"cyrillic"
	"iso-8859-5"
	"iso-ir-144"
	"iso8859-5"
	"iso88595"
	"iso_8859-5"
	"iso_8859-5:1988"
ISO-8859-6	"arabic"
	"asmo-708"
	"csiso88596e"
	"csiso88596i"
	"csisolatinarabic"
	"ecma-114"
	"iso-8859-6"
	"iso-8859-6-e"
	"iso-8859-6-i"
	"iso-ir-127"
	"iso8859-6"
	"iso88596"
	"iso_8859-6"
	"iso_8859-6:1987"
ISO-8859-7	"csisolatingreek"
	"ecma-118"
	"elot_928"
	"greek"
	"greek8"
	"iso-8859-7"
	"iso-ir-126"
	"iso8859-7"
	"iso88597"
	"iso_8859-7"
	"iso_8859-7:1987"
	"sun_eu_greek"
ISO-8859-8	"csiso88598e"
	"csisolatinhebrew"
	"hebrew"
	"iso-8859-8"
	"iso-8859-8-e"
	"iso-ir-138"
	"iso8859-8"
	"iso88598"
	"iso_8859-8"
	"iso_8859-8:1988"
	"visual"
ISO-8859-8-I	"csiso88598i"
	"iso-8859-8-i"

Name	Labels
	"logical"
ISO-8859-10	"csisolatin6"
	"iso-8859-10"
	"iso-ir-157"
	"iso8859-10"
	"iso885910"
	"l6"
	"latin6"
ISO-8859-13	"iso-8859-13"
	"iso8859-13"
	"iso885913"
ISO-8859-14	"iso-8859-14"
	"iso8859-14"
	"iso885914"
ISO-8859-15	"csisolatin9"
	"iso-8859-15"
	"iso8859-15"
	"iso885915"
	"iso_8859-15"
	"l9"
ISO-8859-16	"iso-8859-16"
KOI8-R	"cskoi8r"
	"koi"
	"koi8"
	"koi8-r"
	"koi8_r"
KOI8-U	"koi8-ru"
	"koi8-u"
macintosh	"csmacintosh"
	"mac"
	"macintosh"
	"x-mac-roman"
windows-874	"dos-874"
	"iso-8859-11"
	"iso8859-11"
	"iso885911"
	"tis-620"
	"windows-874"
windows-1250	"cp1250"
	"windows-1250"
	"x-cp1250"
windows-1251	"cp1251"
	"windows-1251"
	"x-cp1251"
windows-1252	"ansi_x3.4-1968"
	"ascii"
	"cp1252"
	"cp819"

Name	Labels
	"csisolatin1"
	"ibm819"
	"iso-8859-1"
	"iso-ir-100"
	"iso8859-1"
	"iso88591"
	"iso_8859-1"
	"iso_8859-1:1987"
	"l1"
	"latin1"
	"us-ascii"
	"windows-1252"
	"x-cp1252"
windows-1253	"cp1253"
	"windows-1253"
	"x-cp1253"
windows-1254	"cp1254"
	"csisolatin5"
	"iso-8859-9"
	"iso-ir-148"
	"iso8859-9"
	"iso88599"
	"iso_8859-9"
	"iso_8859-9:1989"
	"l5"
	"latin5"
	"windows-1254"
	"x-cp1254"
windows-1255	"cp1255"
	"windows-1255"
	"x-cp1255"
windows-1256	"cp1256"
	"windows-1256"
	"x-cp1256"
windows-1257	"cp1257"
	"windows-1257"
	"x-cp1257"
windows-1258	"cp1258"
	"windows-1258"
	"x-cp1258"
x-mac-cyrillic	"x-mac-cyrillic"
	"x-mac-ukrainian"
Legacy multi-byte Chinese (simplified) encodings	
GBK	"chinese"
	"csgb2312"
	"csiso58gb231280"
	"gb2312"

Name	Labels
	"gb_2312"
	"gb_2312-80"
	"gbk"
	"iso-ir-58"
	"x-gbk"
gb18030	"gb18030"
Legacy multi-byte Chinese (traditional) encodings	
Big5	"big5"
	"big5-hkscs"
	"cn-big5"
	"csbig5"
	"x-x-big5"
Legacy multi-byte Japanese encodings	
EUC-JP	"cseucpkdfmtjapanese"
	"euc-jp"
	"x-euc-jp"
ISO-2022-JP	"csiso2022jp"
	"iso-2022-jp"
Shift_JIS	"csshiftjis"
	"ms932"
	"ms_kanji"
	"shift-jis"
	"shift_jis"
	"sjis"
	"windows-31j"
	"x-sjis"
Legacy multi-byte Korean encodings	
EUC-KR	"cseuckr"
	"csksc56011987"
	"euc-kr"
	"iso-ir-149"
	"korean"
	"ks_c_5601-1987"
	"ks_c_5601-1989"
	"ksc5601"
	"ksc_5601"
	"windows-949"
Legacy miscellaneous encodings	
replacement	"csiso2022kr"
	"hz-gb-2312"
	"iso-2022-cn"
	"iso-2022-cn-ext"
	"iso-2022-kr"
	"replacement"
UTF-16BE	"unicodefffe"
	"utf-16be"
UTF-16LE	"csunicode"

Name	Labels
	"iso-10646-ucs-2"
	"ucs-2"
	"unicode"
	"unicodefeff"
	"utf-16"
	"utf-16le"
x-user-defined	"x-user-defined"

Note

All encodings and their labels are also available as non-normative encodings.json resource.

¶ Note

The set of supported encodings is primarily based on the intersection of the sets supported by major browser engines when the development of this standard started, while removing encodings that were rarely used legitimately but that could be used in attacks. The inclusion of some encodings is questionable in the light of anecdotal evidence of the level of use by existing Web content. That is, while they have been broadly supported by browsers, it is unclear if they are broadly used by Web content. However, an effort has not been made to eagerly remove single-byte encodings that were broadly supported by browsers or are part of the ISO 8859 series. In particular, the necessity of the inclusion of IBM866, macintosh, x-mac-cyrillic, ISO-8859-3, ISO-8859-10, ISO-8859-14, and ISO-8859-16 is doubtful for the purpose of supporting existing content, but there are no plans to remove these.

§ 4.3. Output encodings

To **get an output encoding** from an encoding *encoding*, run these steps:

1. If *encoding* is replacement or UTF-16BE/LE, then return UTF-8.
2. Return *encoding*.

Note

The get an output encoding algorithm is useful for URL parsing and HTML form submission, which both need exactly this.

§ 5. Indexes

Most legacy encodings make use of an **index**. An index is an ordered list of entries, each entry consisting of a pointer and a corresponding code point. Within an index pointers are unique and code points can be duplicated.

Note

An efficient implementation likely has two indexes per encoding. One optimized for its decoder and one for its encoder.

To find the pointers and their corresponding code points in an index, let *lines* be the result of splitting the resource's contents on U+000A. Then remove each item in *lines* that is the empty string or starts with U+0023. Then the pointers and their corresponding code points are found by splitting each item in *lines* on U+0009. The first subitem is the pointer (as a decimal number) and the second is the corresponding code point (as a hexadecimal number). Other subitems are not relevant.

Note

To signify changes an index includes an Identifier *and a* Date*. If an* Identifier *has changed, so has the index.*

The **index code point** for *pointer* in *index* is the code point corresponding to *pointer* in *index*, or null if *pointer* is not in *index*.

The **index pointer** for *code point* in *index* is the *first* pointer corresponding to *code point* in *index*, or null if *code point* is not in *index*.

¶ Note

There is a non-normative visualization for each index other than index gb18030 ranges and index ISO-2022-JP katakana. index jis0208 also has an alternative Shift_JIS visualization. Additionally, there is visualization of the Basic Multilingual Plane coverage of each index other than index gb18030 ranges and index ISO-2022-JP katakana.

The legend for the visualizations is:

- *Unmapped*
- *Two bytes in UTF-8*
- *Two bytes in UTF-8, code point follows immediately the code point of previous pointer*
- *Three bytes in UTF-8 (non-PUA)*
- *Three bytes in UTF-8 (non-PUA), code point follows immediately the code point of previous pointer*
- *Private Use*
- *Private Use, code point follows immediately the code point of previous pointer*

- *Four bytes in UTF-8*
- *Four bytes in UTF-8, code point follows immediately the code point of previous pointer*
- *Duplicate code point already mapped at an earlier index*
- *CJK Compatibility Ideograph*
- *CJK Unified Ideographs Extension A*

These are the indexes defined by this specification, excluding index single-byte, which have their own table:

Index				**Notes**
index Big5	index-big5.txt	index Big5 visualization	index Big5 BMP coverage	This matches the Big5 standard in combination with the Hong Kong Supplementary Character Set and other common extensions.
index EUC-KR	index-euc-kr.txt	index EUC-KR visualization	index EUC-KR BMP coverage	This matches the KS X 1001 standard and the Unified Hangul Code, more commonly known together as Windows Codepage 949. It covers the Hangul Syllables block of Unicode in its entirety. The Hangul block whose top left corner in the visualization is at pointer 9026 is in the Unicode order. Taken separately, the rest of the Hangul syllables in this index are in the Unicode order, too.
index gb18030	index-gb18030.txt	index gb18030 visualization	index gb18030 BMP coverage	This matches the GB18030-2005 standard for code points encoded as two bytes, except for 0xA3 0xA0 which maps to U+3000 to be compatible with deployed content. This index covers the CJK Unified Ideographs block of Unicode in its entirety. Entries from that block that are above or to the left of (the first) U+3000 in the visualization are in the Unicode order.
index gb18030 ranges	index-gb18030-ranges.txt			This index works different from all others. Listing all code points would result in over a million items whereas they can be represented neatly in 207 ranges combined with trivial limit checks. It therefore only superficially matches the GB18030-2005 standard for code points encoded as four bytes. See also index gb18030 ranges code point and index gb18030 ranges pointer below.
index jis0208	index-jis0208.txt	index jis0208 visualization, Shift_JIS visualization	index jis0208 BMP coverage	This is the JIS X 0208 standard including formerly proprietary extensions from IBM and NEC.
index jis0212	index-jis0212.txt	index jis0212 visualization	index jis0212 BMP coverage	This is the JIS X 0212 standard. It is only used by the EUC-JP decoder due to lack of widespread support elsewhere.

index ISO-2022-JP katakana	index-iso-2022-jp-katakana.txt	This maps halfwidth to fullwidth katakana as per Unicode Normalization Form KC, except that U+FF9E and U+FF9F map to U+309B and U+309C rather than U+3099 and U+309A. It is only used by the ISO-2022-JP encoder. [UNICODE]

The **index gb18030 ranges code point** for *pointer* is the return value of these steps:

1. If *pointer* is greater than 39419 and less than 189000, or *pointer* is greater than 1237575, return null.
2. If *pointer* is 7457, return code point U+E7C7.
3. Let *offset* be the last pointer in index gb18030 ranges that is less than or equal to *pointer* and let *code point offset* be its corresponding code point.
4. Return a code point whose value is *code point offset* + *pointer* − *offset*.

The **index gb18030 ranges pointer** for *code point* is the return value of these steps:

1. If *code point* is U+E7C7, return pointer 7457.
2. Let *offset* be the last code point in index gb18030 ranges that is less than or equal to *code point* and let *pointer offset* be its corresponding pointer.
3. Return a pointer whose value is *pointer offset* + *code point* − *offset*.

The **index Shift_JIS pointer** for *code point* is the return value of these steps:

1. Let *index* be index jis0208 excluding all entries whose pointer is in the range 8272 to 8835, inclusive.

   Note

   The index jis0208 contains duplicate code points so the exclusion of these entries causes later code points to be used.

2. Return the index pointer for *code point* in *index*.

The **index Big5 pointer** for *code point* is the return value of these steps:

1. Let *index* be index Big5 excluding all entries whose pointer is less than (0xA1 - 0x81) × 157.

   Note

   Avoid returning Hong Kong Supplementary Character Set extensions literally.

2. If *code point* is U+2550, U+255E, U+2561, U+256A, U+5341, or U+5345, return the *last* pointer corresponding to *code point* in *index*.

   Note

   There are other duplicate code points, but for those the first pointer is to be used.

3. Return the index pointer for *code point* in *index*.

Note

All indexes are also available as a non-normative indexes.json resource. (Index gb18030 ranges has a slightly different format here, to be able to represent ranges.)

§ 6. Hooks for standards

Note

The algorithms defined below (UTF-8 decode, UTF-8 decode without BOM, UTF-8 decode without BOM or fail, and UTF-8 encode) are intended for usage by other standards.

For decoding, UTF-8 decode is to be used by new formats. For identifiers or byte sequences within a format or protocol, use UTF-8 decode without BOM or UTF-8 decode without BOM or fail.

For encoding, UTF-8 encode is to be used.

Standards are to ensure that the input I/O queues they pass to UTF-8 encode (as well as the legacy encode) are effectively I/O queues of scalar values, i.e., they contain no surrogates.

These hooks (as well as decode and encode) will block until the input I/O queue has been consumed in its entirety. In order to use the output tokens as they are pushed into the stream, callers are to invoke the hooks with an empty output I/O queue and read from it in parallel. Note that some care is needed when using UTF-8 decode without BOM or fail, as any error found during decoding will prevent the end-of-queue item from ever being pushed into the output I/O queue.

To **UTF-8 decode** an I/O queue of bytes *ioQueue* given an optional I/O queue of scalar values *output* (default « »), run these steps:

1. Let *buffer* be the result of peeking three bytes from *ioQueue*, converted to a byte sequence.
2. If *buffer* is 0xEF 0xBB 0xBF, then read three bytes from *ioQueue*. (Do nothing with those bytes.)
3. Process a queue with an instance of UTF-8's decoder, *ioQueue*, *output*, and `"replacement"`.
4. Return *output*.

To **UTF-8 decode without BOM** an I/O queue of bytes *ioQueue* given an optional I/O queue of scalar values *output* (default « »), run these steps:

1. Process a queue with an instance of UTF-8's decoder, *ioQueue*, *output*, and `"replacement"`.
2. Return *output*.

To **UTF-8 decode without BOM or fail** an I/O queue of bytes *ioQueue* given an optional I/O queue of scalar values *output* (default « »), run these steps:

1. Let *potentialError* be the result of processing a queue with an instance of UTF-8's decoder, *ioQueue*, *output*, and `"fatal"`.
2. If *potentialError* is an error, then return failure.
3. Return *output*.

To **UTF-8 encode** an I/O queue of scalar values *ioQueue* given an optional I/O queue of bytes *output* (default « »), return the result of encoding *ioQueue* with encoding UTF-8 and *output*.

§ 6.1. Legacy hooks for standards

Note

Standards are strongly discouraged from using decode, BOM sniff, and encode, except as needed for compatibility. Standards needing these legacy hooks will most likely also need to use get an encoding (to turn a label into an encoding) and get an output encoding (to turn an encoding into another encoding that is suitable to pass into encode).

For the extremely niche case of URL percent-encoding, custom encoder error handling is needed. The get an encoder and encode or fail algorithms are to be used for that. Other algorithms are not to be used directly.

To **decode** an I/O queue of bytes *ioQueue* given a fallback encoding *encoding* and an optional I/O queue of scalar values *output* (default « »), run these steps:

1. Let *BOMEncoding* be the result of BOM sniffing *ioQueue*.
2. If *BOMEncoding* is non-null:
    1. Set *encoding* to *BOMEncoding*.
    2. Read three bytes from *ioQueue*, if *BOMEncoding* is UTF-8; otherwise read two bytes. (Do nothing with those bytes.)

    Note

    For compatibility with deployed content, the byte order mark is more authoritative than anything else. In a context where HTTP is used this is in violation of the semantics of the `` `Content-Type` `` *header.*
3. Process a queue with an instance of *encoding*'s decoder, *ioQueue*, *output*, and `"replacement"`.
4. Return *output*.

To **BOM sniff** an I/O queue of bytes *ioQueue*, run these steps:

1. Let *BOM* be the result of peeking 3 bytes from *ioQueue*, converted to a byte sequence.
2. For each of the rows in the table below, starting with the first one and going down, if *BOM* starts with the bytes given in the first column, then return the encoding given in the cell in the second column of that row. Otherwise, return null.

Byte order mark	Encoding
0xEF 0xBB 0xBF	UTF-8
0xFE 0xFF	UTF-16BE
0xFF 0xFE	UTF-16LE

Note

This hook is a workaround for the fact that decode has no way to communicate back to the caller that it has found a byte order mark and is therefore not using the provided encoding. The hook is to be invoked before decode, and it will return an encoding corresponding to the byte order mark found, or null otherwise.

To **encode** an I/O queue of scalar values *ioQueue* given an encoding *encoding* and an optional I/O queue of bytes *output* (default « »), run these steps:

1. Let *encoder* be the result of getting an encoder from *encoding*.
2. Process a queue with *encoder*, *ioQueue*, *output*, and "`html`".
3. Return *output*.

Note

This is a legacy hook for HTML forms. Layering UTF-8 encode on top is safe as it never triggers errors. [HTML]

To **get an encoder** from an encoding *encoding*:

1. Assert: *encoding* is not replacement or UTF-16BE/LE.
2. Return an instance of *encoding*'s encoder.

To **encode or fail** an I/O queue of scalar values *ioQueue* given an encoder instance *encoder* and an I/O queue of bytes *output*, run these steps:

1. Let *potentialError* be the result of processing a queue with *encoder*, *ioQueue*, *output*, and "`fatal`".
2. Push end-of-queue to *output*.
3. If *potentialError* is an error, then return error's code point's value.
4. Return null.

¶ Note

This is a legacy hook for URL percent-encoding. The caller will have to keep an encoder instance alive as the ISO-2022-JP encoder can be in two different states when returning an error. That also means that if the caller emits bytes to encode the error in some way, these have to be in the range 0x00 to 0x7F, inclusive, excluding 0x0E, 0x0F, 0x1B, 0x5C, and 0x7E. [URL]

In particular, if upon returning an error the ISO-2022-JP encoder is in the Roman state, the caller cannot output 0x5C (\) as it will not decode as U+005C (\). For this reason, applications using encode or fail for unintended purposes ought to take care to prevent the use of the ISO-2022-JP encoder in combination with replacement schemes, such as those of JavaScript and CSS, that use U+005C (\) as part of the replacement syntax (e.g., `\u2603`*) or make sure to pass the replacement syntax through the encoder (in contrast to URL percent-encoding).*

The return value is either the number representing the code point that could not be encoded or null, if there was no error. When it returns non-null the caller will have to invoke it again, supplying the same encoder instance and a new output

I/O queue.

§ 7. API

This section uses terminology from Web IDL. Browser user agents must support this API. JavaScript implementations should support this API. Other user agents or programming languages are encouraged to use an API suitable to their needs, which might not be this one. [WEBIDL]

¶ Example

The following example uses the TextEncoder object to encode an array of strings into an ArrayBuffer. The result is a Uint8Array containing the number of strings (as a Uint32Array), followed by the length of the first string (as a Uint32Array), the UTF-8 encoded string data, the length of the second string (as a Uint32Array), the string data, and so on.

```
function encodeArrayOfStrings(strings) {
  var encoder, encoded, len, bytes, view, offset;

  encoder = new TextEncoder();
  encoded = [];

  len = Uint32Array.BYTES_PER_ELEMENT;
  for (var i = 0; i < strings.length; i++) {
    len += Uint32Array.BYTES_PER_ELEMENT;
    encoded[i] = encoder.encode(strings[i]);
    len += encoded[i].byteLength;
  }

  bytes = new Uint8Array(len);
  view = new DataView(bytes.buffer);
  offset = 0;

  view.setUint32(offset, strings.length);
  offset += Uint32Array.BYTES_PER_ELEMENT;
  for (var i = 0; i < encoded.length; i += 1) {
    len = encoded[i].byteLength;
    view.setUint32(offset, len);
    offset += Uint32Array.BYTES_PER_ELEMENT;
    bytes.set(encoded[i], offset);
    offset += len;
  }
  return bytes.buffer;
}
```

The following example decodes an ArrayBuffer containing data encoded in the format produced by the previous example, or an equivalent algorithm for encodings other than UTF-8, back into an array of strings.

```
function decodeArrayOfStrings(buffer, encoding) {
  var decoder, view, offset, num_strings, strings, len;

  decoder = new TextDecoder(encoding);
  view = new DataView(buffer);
  offset = 0;
  strings = [];
```

```
    num_strings = view.getUint32(offset);
    offset += Uint32Array.BYTES_PER_ELEMENT;
    for (var i = 0; i < num_strings; i++) {
      len = view.getUint32(offset);
      offset += Uint32Array.BYTES_PER_ELEMENT;
      strings[i] = decoder.decode(
        new DataView(view.buffer, offset, len));
      offset += len;
    }
    return strings;
  }
```

§ 7.1. Interface mixin TextDecoderCommon

```
interface mixin TextDecoderCommon {
  readonly attribute DOMString encoding;
  readonly attribute boolean fatal;
  readonly attribute boolean ignoreBOM;
};
```

The TextDecoderCommon interface mixin defines common getters that are shared between TextDecoder and TextDecoderStream objects. These objects have an associated:

encoding
: An encoding.

decoder
: A decoder instance.

I/O queue
: An I/O queue of bytes.

ignore BOM
: A boolean, initially false.

BOM seen
: A boolean, initially false.

error mode
: An error mode, initially "replacement".

The **serialize I/O queue** algorithm, given a TextDecoderCommon *decoder* and an I/O queue of scalar values *ioQueue*, runs these steps:

1. Let *output* be the empty string.
2. While true:
    1. Let *item* be the result of reading from *ioQueue*.
    2. If *item* is end-of-queue, then return *output*.
    3. If *decoder*'s encoding is UTF-8 or UTF-16BE/LE, and *decoder*'s ignore

BOM and BOM seen are false, then:

1. Set *decoder*’s BOM seen to true.

2. If *item* is U+FEFF, then continue.

4. Append *item* to *output*.

Note

This algorithm is intentionally different with respect to BOM handling from the decode algorithm used by the rest of the platform to give API users more control.

The **`encoding`** getter steps are to return this’s encoding’s name, ASCII lowercased.

The **`fatal`** getter steps are to return true if this’s error mode is "`fatal`", otherwise false.

The **`ignoreBOM`** getter steps are to return this’s ignore BOM.

§ 7.2. Interface `TextDecoder`

```
dictionary TextDecoderOptions {
  boolean fatal = false;
  boolean ignoreBOM = false;
};

dictionary TextDecodeOptions {
  boolean stream = false;
};

[Exposed=(Window,Worker)]
interface TextDecoder {
  constructor(optional DOMString label = "utf-8", optional
TextDecoderOptions options = {});

  USVString decode(optional [AllowShared] BufferSource input,
optional TextDecodeOptions options = {});
};
TextDecoder includes TextDecoderCommon;
```

A `TextDecoder` object has an associated **do not flush**, which is a boolean, initially false.

For web developers (non-normative)

***decoder* = new TextDecoder([*label* = "utf-8" [, *options*]])**

Returns a new `TextDecoder` object.

If *label* is either not a label or is a label for replacement, throws a `RangeError`.

***decoder* . encoding**

Returns encoding’s name, lowercased.

`decoder . fatal`

Returns true if error mode is "`fatal`", otherwise false.

`decoder . ignoreBOM`

Returns the value of ignore BOM.

`decoder . decode([input [, options]])`

Returns the result of running encoding's decoder. The method can be invoked zero or more times with *options*'s `stream` set to true, and then once without *options*'s `stream` (or set to false), to process a fragmented input. If the invocation without *options*'s `stream` (or set to false) has no *input*, it's clearest to omit both arguments.

¶ Example

```
var string = "", decoder = new TextDecoder(encoding),
buffer;
while(buffer = next_chunk()) {
  string += decoder.decode(buffer, {stream:true});
}
string += decoder.decode(); // end-of-queue
```

If the error mode is "`fatal`" and encoding's decoder returns error, throws a `TypeError`.

The **`new TextDecoder(label, options)`** constructor steps are:

1. Let *encoding* be the result of getting an encoding from *label*.
2. If *encoding* is failure or replacement, then throw a `RangeError`.
3. Set this's encoding to *encoding*.
4. If *options*["`fatal`"] is true, then set this's error mode to "`fatal`".
5. Set this's ignore BOM to *options*["`ignoreBOM`"].

The **`decode(input, options)`** method steps are:

1. If this's do not flush is false, then set this's decoder to a new instance of this's encoding's decoder, this's I/O queue to the I/O queue of bytes « end-of-queue », and this's BOM seen to false.
2. Set this's do not flush to *options*["`stream`"].
3. If *input* is given, then push a copy of *input* to this's I/O queue.

Note

Implementations are strongly encouraged to use an implementation strategy that avoids this copy. When doing so they will have to make sure that changes to input *do not affect future calls to* `decode()`.

⚠Warning!

The memory exposed by **`SharedArrayBuffer`** ***objects does not adhere to data race freedom properties required by the memory model of programming languages typically used for implementations. When implementing, take care to use the***

appropriate facilities when accessing memory exposed by `SharedArrayBuffer` ***objects.***

4. Let *output* be the I/O queue of scalar values « end-of-queue ».

5. While true:

    1. Let *item* be the result of reading from this's I/O queue.

    2. If *item* is end-of-queue and this's do not flush is true, then return the result of running serialize I/O queue with this and *output*.

       Note

       The way streaming works is to not handle end-of-queue here when this's do not flush is true and to not set it to false. That way in a subsequent invocation this's decoder is not set anew in the first step of the algorithm and its state is preserved.

    3. Otherwise:

        1. Let *result* be the result of processing an item with *item*, this's decoder, this's I/O queue, *output*, and this's error mode.

        2. If *result* is finished, then return the result of running serialize I/O queue with this and *output*.

        3. Otherwise, if *result* is error, throw a `TypeError`.

§ 7.3. Interface mixin `TextEncoderCommon`

```
interface mixin TextEncoderCommon {
  readonly attribute DOMString encoding;
};
```

The `TextEncoderCommon` interface mixin defines common getters that are shared between `TextEncoder` and `TextEncoderStream` objects.

The **`encoding`** getter steps are to return "`utf-8`".

§ 7.4. Interface `TextEncoder`

```
dictionary TextEncoderEncodeIntoResult {
  unsigned long long read;
  unsigned long long written;
};

[Exposed=(Window,Worker)]
interface TextEncoder {
  constructor();
```

```
  [NewObject] Uint8Array encode(optional USVString input =
"");
  TextEncoderEncodeIntoResult encodeInto(USVString source,
[AllowShared] Uint8Array destination);
};
TextEncoder includes TextEncoderCommon;
```

Note

A `TextEncoder` object offers no label *argument as it only supports UTF-8. It also offers no* `stream` *option as no encoder requires buffering of scalar values.*

For web developers (non-normative)

encoder **= new TextEncoder()**

Returns a new `TextEncoder` object.

encoder **. encoding**

Returns "`utf-8`".

encoder **. encode([*input* = ""])**

Returns the result of running UTF-8's encoder.

encoder **. encodeInto(*source*, *destination*)**

Runs the UTF-8 encoder on *source*, stores the result of that operation into *destination*, and returns the progress made as an object wherein `read` is the number of converted code units of *source* and `written` is the number of bytes modified in *destination*.

The **`new TextEncoder()`** constructor steps are to do nothing.

The **`encode(input)`** method steps are:

1. Convert *input* to an I/O queue of scalar values.
2. Let *output* be the I/O queue of bytes « end-of-queue ».
3. While true:
    1. Let *item* be the result of reading from *input*.
    2. Let *result* be the result of processing an item with *item*, an instance of the UTF-8 encoder, *input*, *output*, and "`fatal`".
    3. Assert: *result* is not an error.

       Note

       The UTF-8 encoder cannot return error.

    4. If *result* is finished, then convert *output* into a byte sequence and return a `Uint8Array` object wrapping an `ArrayBuffer` containing *output*.

The **`encodeInto(source, destination)`** method steps are:

1. Let *read* be 0.
2. Let *written* be 0.

3. Let *destinationBytes* be the result of getting a reference to the bytes held by *destination*.

4. Let *encoder* be an instance of the UTF-8 encoder.

5. Let *unused* be the I/O queue of scalar values « end-of-queue ».

   Note

   The handler algorithm invoked below requires this argument, but it is not used by the UTF-8 encoder.

6. Convert *source* to an I/O queue of scalar values.

7. While true:

   1. Let *item* be the result of reading from *source*.

   2. Let *result* be the result of running *encoder*'s handler on *unused* and *item*.

   3. If *result* is finished, then break.

   4. Otherwise:

      1. If *destinationBytes*'s length − *written* is greater than or equal to the number of bytes in *result*, then:

         1. If *item* is greater than U+FFFF, then increment *read* by 2.

         2. Otherwise, increment *read* by 1.

         3. Write the bytes in *result* into *destinationBytes*, from byte offset *written*.

            ⚠Warning!

            See the warning for `SharedArrayBuffer` objects above.

         4. Increment *written* by the number of bytes in *result*.

      2. Otherwise, break.

8. Return «["`read`" → *read*, "`written`" → *written*]».

¶ Example

The encodeInto() method can be used to encode a string into an existing `ArrayBuffer` object. Various details below are left as an exercise for the reader, but this demonstrates an approach one could take to use this method:

```
function convertString(buffer, input, callback) {
  let bufferSize = 256,
      bufferStart = malloc(buffer, bufferSize),
      writeOffset = 0,
      readOffset = 0;
  while (true) {
    const view = new Uint8Array(buffer, bufferStart +
writeOffset, bufferSize - writeOffset),
          {read, written} =
```

```
cachedEncoder.encodeInto(input.substring(readOffset),
view);
    readOffset += read;
    writeOffset += written;
    if (readOffset === input.length) {
      callback(bufferStart, writeOffset);
      free(buffer, bufferStart);
      return;
    }
    bufferSize *= 2;
    bufferStart = realloc(buffer, bufferStart, bufferSize);
  }
}
```

§ 7.5. Interface TextDecoderStream

```
[Exposed=(Window,Worker)]
interface TextDecoderStream {
  constructor(optional DOMString label = "utf-8", optional
TextDecoderOptions options = {});
};
TextDecoderStream includes TextDecoderCommon;
TextDecoderStream includes GenericTransformStream;
```

For web developers (non-normative)

***decoder* = new TextDecoderStream([*label* = "utf-8" [, *options*]])**

Returns a new `TextDecoderStream` object.

If *label* is either not a label or is a label for replacement, throws a `RangeError`.

***decoder* . encoding**

Returns encoding's name, lowercased.

***decoder* . fatal**

Returns true if error mode is "`fatal`", and false otherwise.

***decoder* . ignoreBOM**

Returns the value of ignore BOM.

***decoder* . readable**

Returns a readable stream whose chunks are strings resulting from running encoding's decoder on the chunks written to `writable`.

***decoder* . writable**

Returns a writable stream which accepts [`AllowShared`] `BufferSource` chunks and runs them through encoding's decoder before making them available to `readable`.

Typically this will be used via the `pipeThrough()` method on a `ReadableStream` source.

¶ Example

```
var decoder = new TextDecoderStream(encoding);
byteReadable
  .pipeThrough(decoder)
  .pipeTo(textWritable);
```

If the error mode is "fatal" and encoding's decoder returns error, both readable and writable will be errored with a TypeError.

The **new TextDecoderStream(*label*, *options*)** constructor steps are:

1. Let *encoding* be the result of getting an encoding from *label*.
2. If *encoding* is failure or replacement, then throw a RangeError.
3. Set this's encoding to *encoding*.
4. If *options*["fatal"] is true, then set this's error mode to "fatal".
5. set this's ignore BOM to *options*["ignoreBOM"].
6. Set this's decoder to a new instance of this's encoding's decoder, and set this's I/O queue to a new I/O queue.
7. Let *transformAlgorithm* be an algorithm which takes a *chunk* argument and runs the decode and enqueue a chunk algorithm with this and *chunk*.
8. Let *flushAlgorithm* be an algorithm which takes no arguments and runs the flush and enqueue algorithm with this.
9. Set this's transform to the result of creating a TransformStream with *transformAlgorithm* set to *transformAlgorithm* and *flushAlgorithm* set to *flushAlgorithm*.

The **decode and enqueue a chunk** algorithm, given a TextDecoderStream object *decoder* and a *chunk*, runs these steps:

1. Let *bufferSource* be the result of converting *chunk* to an [AllowShared] BufferSource.
2. Push a copy of *bufferSource* to *decoder*'s I/O queue.

   ⚠Warning!

   See the warning for SharedArrayBuffer objects above.

3. Let *output* be the I/O queue of scalar values « end-of-queue ».
4. While true:
   1. Let *item* be the result of reading from *decoder*'s I/O queue.
   2. If *item* is end-of-queue, then:
      1. Let *outputChunk* be the result of running serialize I/O queue with *decoder* and *output*.
      2. If *outputChunk* is non-empty, then enqueue *outputChunk* in *decoder*'s transform.

3. Return.

3. Let *result* be the result of processing an item with *item*, *decoder*'s decoder, *decoder*'s I/O queue, *output*, and *decoder*'s error mode.

4. If *result* is error, then throw a `TypeError`.

The **flush and enqueue** algorithm, which handles the end of data from the input `ReadableStream` object, given a `TextDecoderStream` object *decoder*, runs these steps:

1. Let *output* be the I/O queue of scalar values « end-of-queue ».

2. Let *result* be the result of processing an item with end-of-queue, *decoder*'s decoder, *decoder*'s I/O queue, *output*, and *decoder*'s error mode.

3. If *result* is finished, then:

    1. Let *outputChunk* be the result of running serialize I/O queue with *decoder* and *output*.

    2. If *outputChunk* is non-empty, then enqueue *outputChunk* in *decoder*'s transform.

4. Otherwise, throw a `TypeError`.

§ 7.6. Interface `TextEncoderStream`

```
[Exposed=(Window,Worker)]
interface TextEncoderStream {
  constructor();
};
TextEncoderStream includes TextEncoderCommon;
TextEncoderStream includes GenericTransformStream;
```

A `TextEncoderStream` object has an associated:

encoder
: An encoder instance.

pending high surrogate
: Null or a surrogate, initially null.

Note

A `TextEncoderStream` object offers no label *argument as it only supports UTF-8.*

For web developers (non-normative)

***encoder* = new TextEncoderStream()**
: Returns a new `TextEncoderStream` object.

***encoder* . encoding**
: Returns "`utf-8`".

***encoder* . readable**
: Returns a readable stream whose chunks are `Uint8Array`s resulting from

running UTF-8's encoder on the chunks written to `writable`.

***encoder* . `writable`**

Returns a writable stream which accepts string chunks and runs them through UTF-8's encoder before making them available to `readable`.

Typically this will be used via the `pipeThrough()` method on a `ReadableStream` source.

¶ Example

```
textReadable
  .pipeThrough(new TextEncoderStream())
  .pipeTo(byteWritable);
```

The **`new TextEncoderStream()`** constructor steps are:

1. Set this's encoder to an instance of the UTF-8 encoder.
2. Let *transformAlgorithm* be an algorithm which takes a *chunk* argument and runs the encode and enqueue a chunk algorithm with this and *chunk*.
3. Let *flushAlgorithm* be an algorithm which runs the encode and flush algorithm with this.
4. Set this's transform to the result of creating a `TransformStream` with *transformAlgorithm* set to *transformAlgorithm* and *flushAlgorithm* set to *flushAlgorithm*.

The **encode and enqueue a chunk** algorithm, given a `TextEncoderStream` object *encoder* and *chunk*, runs these steps:

1. Let *input* be the result of converting *chunk* to a `DOMString`.
2. Convert *input* to an I/O queue of code units.

   Note

   `DOMString`, as well as an I/O queue of code units rather than scalar values, are used here so that a surrogate pair that is split between chunks can be reassembled into the appropriate scalar value. The behavior is otherwise identical to `USVString`. In particular, lone surrogates will be replaced with U+FFFD.

3. Let *output* be the I/O queue of bytes « end-of-queue ».
4. While true:
   1. Let *item* be the result of reading from *input*.
   2. If *item* is end-of-queue, then:
      1. Convert *output* into a byte sequence.
      2. If *output* is non-empty, then:
         1. Let *chunk* be a `Uint8Array` object wrapping an `ArrayBuffer` containing *output*.

2. Enqueue *chunk* into *encoder*'s transform.

3. Return.

3. Let *result* be the result of executing the convert code unit to scalar value algorithm with *encoder*, *item* and *input*.

4. If *result* is not continue, then process an item with *result*, *encoder*'s encoder, *input*, *output*, and "`fatal`".

The **convert code unit to scalar value** algorithm, given a `TextEncoderStream` object *encoder*, a code unit *item*, and an I/O queue of code units *input*, runs these steps:

1. If *encoder*'s pending high surrogate is non-null, then:
    1. Let *high surrogate* be *encoder*'s pending high surrogate.
    2. Set *encoder*'s pending high surrogate to null.
    3. If *item* is in the range U+DC00 to U+DFFF, inclusive, then return a scalar value whose value is 0x10000 + ((*high surrogate* − 0xD800) << 10) + (*item* − 0xDC00).
    4. Prepend *item* to *input*.
    5. Return U+FFFD.
2. If *item* is in the range U+D800 to U+DBFF, inclusive, then set pending high surrogate to *item* and return continue.
3. If *item* is in the range U+DC00 to U+DFFF, inclusive, then return U+FFFD.
4. Return *item*.

Note

This is equivalent to the "convert a string into a scalar value string" algorithm from the Infra Standard, but allows for surrogate pairs that are split between strings. [INFRA]

The **encode and flush** algorithm, given a `TextEncoderStream` object *encoder*, runs these steps:

1. If *encoder*'s pending high surrogate is non-null, then:
    1. Let *chunk* be a `Uint8Array` object wrapping an `ArrayBuffer` containing 0xEF 0xBF 0xBD.

       Note

       This is U+FFFD (�) in UTF-8 bytes.

    2. Enqueue *chunk* into *encoder*'s transform.

§ 8. The encoding

§ 8.1. UTF-8

§ 8.1.1. UTF-8 decoder

Note

A byte order mark has priority over a label as it has been found to be more accurate in deployed content. Therefore it is not part of the UTF-8 decoder algorithm but rather the decode and UTF-8 decode algorithms.

UTF-8's decoder has an associated **UTF-8 code point**, **UTF-8 bytes seen**, and **UTF-8 bytes needed** (all initially 0), a **UTF-8 lower boundary** (initially 0x80), and a **UTF-8 upper boundary** (initially 0xBF).

UTF-8's decoder's handler, given *ioQueue* and *byte*, runs these steps:

1. If *byte* is end-of-queue and UTF-8 bytes needed is not 0, set UTF-8 bytes needed to 0 and return error.
2. If *byte* is end-of-queue, return finished.
3. If UTF-8 bytes needed is 0, based on *byte*:

   ↪ **0x00 to 0x7F**

   Return a code point whose value is *byte*.

   ↪ **0xC2 to 0xDF**

   1. Set UTF-8 bytes needed to 1.
   2. Set UTF-8 code point to *byte* & 0x1F.

      Note

      The five least significant bits of byte*.*

   ↪ **0xE0 to 0xEF**

   1. If *byte* is 0xE0, set UTF-8 lower boundary to 0xA0.
   2. If *byte* is 0xED, set UTF-8 upper boundary to 0x9F.
   3. Set UTF-8 bytes needed to 2.
   4. Set UTF-8 code point to *byte* & 0xF.

      Note

      The four least significant bits of byte*.*

   ↪ **0xF0 to 0xF4**

   1. If *byte* is 0xF0, set UTF-8 lower boundary to 0x90.
   2. If *byte* is 0xF4, set UTF-8 upper boundary to 0x8F.
   3. Set UTF-8 bytes needed to 3.
   4. Set UTF-8 code point to *byte* & 0x7.

Note

The three least significant bits of byte.

↪ **Otherwise**

Return error.

Return continue.

4. If *byte* is not in the range UTF-8 lower boundary to UTF-8 upper boundary, inclusive, then:

    1. Set UTF-8 code point, UTF-8 bytes needed, and UTF-8 bytes seen to 0, set UTF-8 lower boundary to 0x80, and set UTF-8 upper boundary to 0xBF.

    2. Prepend *byte* to *ioQueue*.

    3. Return error.

5. Set UTF-8 lower boundary to 0x80 and UTF-8 upper boundary to 0xBF.

6. Set UTF-8 code point to (UTF-8 code point << 6) | (*byte* & 0x3F)

    Note

    Shift the existing bits of UTF-8 code point left by six places and set the newly-vacated six least significant bits to the six least significant bits of byte.

7. Increase UTF-8 bytes seen by one.

8. If UTF-8 bytes seen is not equal to UTF-8 bytes needed, return continue.

9. Let *code point* be UTF-8 code point.

10. Set UTF-8 code point, UTF-8 bytes needed, and UTF-8 bytes seen to 0.

11. Return a code point whose value is *code point*.

Note

The constraints in the UTF-8 decoder above match "Best Practices for Using U+FFFD" from the Unicode standard. No other behavior is permitted per the Encoding Standard (other algorithms that achieve the same result are fine, even encouraged). [UNICODE]

§ 8.1.2. UTF-8 encoder

UTF-8's encoder's handler, given *ioQueue* and *code point*, runs these steps:

1. If *code point* is end-of-queue, return finished.

2. If *code point* is an ASCII code point, return a byte whose value is *code point*.

3. Set *count* and *offset* based on the range *code point* is in:

    ↪ **U+0080 to U+07FF, inclusive**

    1 and 0xC0

↪ **U+0800 to U+FFFF, inclusive**

2 and 0xE0

↪ **U+10000 to U+10FFFF, inclusive**

3 and 0xF0

4. Let *bytes* be a byte sequence whose first byte is (*code point* >> (6 × *count*)) + *offset*.

5. While *count* is greater than 0:

    1. Set *temp* to *code point* >> (6 × (*count* − 1)).

    2. Append to *bytes* 0x80 | (*temp* & 0x3F).

    3. Decrease *count* by one.

6. Return bytes *bytes*, in order.

Note

This algorithm has identical results to the one described in the Unicode standard. It is included here for completeness. [UNICODE]

§ 9. Legacy single-byte encodings

An encoding where each byte is either a single code point or nothing, is a **single-byte encoding**. Single-byte encodings share the decoder and encoder. **Index single-byte**, as referenced by the single-byte decoder and single-byte encoder, is defined by the following table, and depends on the single-byte encoding in use. All but two single-byte encodings have a unique index.

IBM866	index-ibm866.txt	index IBM866 visualization	index IBM866 BMP coverage
ISO-8859-2	index-iso-8859-2.txt	index ISO-8859-2 visualization	index ISO-8859-2 BMP coverage
ISO-8859-3	index-iso-8859-3.txt	index ISO-8859-3 visualization	index ISO-8859-3 BMP coverage
ISO-8859-4	index-iso-8859-4.txt	index ISO-8859-4 visualization	index ISO-8859-4 BMP coverage
ISO-8859-5	index-iso-8859-5.txt	index ISO-8859-5 visualization	index ISO-8859-5 BMP coverage
ISO-8859-6	index-iso-8859-6.txt	index ISO-8859-6 visualization	index ISO-8859-6 BMP coverage
ISO-8859-7	index-iso-8859-7.txt	index ISO-8859-7 visualization	index ISO-8859-7 BMP coverage
ISO-8859-8 **ISO-8859-8-I**	index-iso-8859-8.txt	index ISO-8859-8 visualization	index ISO-8859-8 BMP coverage
ISO-8859-10	index-iso-8859-10.txt	index ISO-8859-10 visualization	index ISO-8859-10 BMP coverage
ISO-8859-13	index-iso-8859-13.txt	index ISO-8859-13 visualization	index ISO-8859-13 BMP coverage
ISO-8859-14	index-iso-8859-14.txt	index ISO-8859-14 visualization	index ISO-8859-14 BMP coverage
ISO-8859-15	index-iso-8859-15.txt	index ISO-8859-15 visualization	index ISO-8859-15 BMP coverage
ISO-8859-16	index-iso-8859-16.txt	index ISO-8859-16 visualization	index ISO-8859-16 BMP coverage
KOI8-R	index-koi8-r.txt	index KOI8-R visualization	index KOI8-R BMP coverage
KOI8-U	index-koi8-u.txt	index KOI8-U visualization	index KOI8-U BMP coverage
macintosh	index-macintosh.txt	index macintosh visualization	index macintosh BMP coverage
windows-874	index-windows-874.txt	index windows-874 visualization	index windows-874 BMP coverage
windows-1250	index-windows-1250.txt	index windows-1250 visualization	index windows-1250 BMP coverage
windows-1251	index-windows-1251.txt	index windows-1251 visualization	index windows-1251 BMP coverage
windows-1252	index-windows-1252.txt	index windows-1252 visualization	index windows-1252 BMP coverage
windows-1253	index-windows-1253.txt	index windows-1253 visualization	index windows-1253 BMP coverage
windows-1254	index-windows-1254.txt	index windows-1254 visualization	index windows-1254 BMP coverage
windows-1255	index-windows-1255.txt	index windows-1255 visualization	index windows-1255 BMP coverage
windows-1256	index-windows-1256.txt	index windows-1256 visualization	index windows-1256 BMP coverage
windows-1257	index-windows-1257.txt	index windows-1257 visualization	index windows-1257 BMP coverage

windows-1258	index-windows-1258.txt	index windows-1258 visualization	index windows-1258 BMP coverage
x-mac-cyrillic	index-x-mac-cyrillic.txt	index x-mac-cyrillic visualization	index x-mac-cyrillic BMP coverage

Note

ISO-8859-8 and ISO-8859-8-I are distinct encoding names, because ISO-8859-8 has influence on the layout direction. And although historically this might have been the case for ISO-8859-6 and "ISO-8859-6-I" as well, that is no longer true.

§ 9.1. single-byte decoder

Single-byte encodings's decoder's handler, given *ioQueue* and *byte*, runs these steps:

1. If *byte* is end-of-queue, return finished.
2. If *byte* is an ASCII byte, return a code point whose value is *byte*.
3. Let *code point* be the index code point for *byte* − 0x80 in index single-byte.
4. If *code point* is null, return error.
5. Return a code point whose value is *code point*.

§ 9.2. single-byte encoder

Single-byte encodings's encoder's handler, given *ioQueue* and *code point*, runs these steps:

1. If *code point* is end-of-queue, return finished.
2. If *code point* is an ASCII code point, return a byte whose value is *code point*.
3. Let *pointer* be the index pointer for *code point* in index single-byte.
4. If *pointer* is null, return error with *code point*.
5. Return a byte whose value is *pointer* + 0x80.

§ 10. Legacy multi-byte Chinese (simplified) encodings

§ 10.1. GBK

§ 10.1.1. GBK decoder

GBK's decoder is gb18030's decoder.

§ 10.1.2. GBK encoder

GBK's encoder is gb18030's encoder with its is GBK set to true.

Note

Not fully aliasing GBK with gb18030 is a conservative move to decrease the chances of breaking legacy servers and other consumers of content generated with GBK's encoder.

§ 10.2. gb18030

§ 10.2.1. gb18030 decoder

gb18030's decoder has an associated **gb18030 first**, **gb18030 second**, and **gb18030 third** (all initially 0x00).

gb18030's decoder's handler, given *ioQueue* and *byte*, runs these steps:

1. If *byte* is end-of-queue and gb18030 first, gb18030 second, and gb18030 third are 0x00, return finished.
2. If *byte* is end-of-queue, and gb18030 first, gb18030 second, or gb18030 third is not 0x00, set gb18030 first, gb18030 second, and gb18030 third to 0x00, and return error.
3. If gb18030 third is not 0x00, then:
    1. If *byte* is not in the range 0x30 to 0x39, inclusive, then:
        1. Prepend gb18030 second, gb18030 third, and *byte* to *ioQueue*.
        2. Set gb18030 first, gb18030 second, and gb18030 third to 0x00.
        3. Return error.
    2. Let *code point* be the index gb18030 ranges code point for ((gb18030 first − 0x81) × (10 × 126 × 10)) + ((gb18030 second − 0x30) × (10 × 126)) + ((gb18030 third − 0x81) × 10) + *byte* − 0x30.
    3. Set gb18030 first, gb18030 second, and gb18030 third to 0x00.
    4. If *code point* is null, return error.
    5. Return a code point whose value is *code point*.
4. If gb18030 second is not 0x00, then:

    1. If *byte* is in the range 0x81 to 0xFE, inclusive, set gb18030 third to *byte* and return continue.
    2. Prepend gb18030 second followed by *byte* to *ioQueue*, set gb18030 first and gb18030 second to 0x00, and return error.
5. If gb18030 first is not 0x00, then:
    1. If *byte* is in the range 0x30 to 0x39, inclusive, set gb18030 second to *byte* and return continue.
    2. Let *lead* be gb18030 first, let *pointer* be null, and set gb18030 first to 0x00.
    3. Let *offset* be 0x40 if *byte* is less than 0x7F, otherwise 0x41.
    4. If *byte* is in the range 0x40 to 0x7E, inclusive, or 0x80 to 0xFE, inclusive, set *pointer* to (*lead* − 0x81) × 190 + (*byte* − *offset*).
    5. Let *code point* be null if *pointer* is null, otherwise the index code point for *pointer* in index gb18030.
    6. If *code point* is non-null, return a code point whose value is *code point*.
    7. If *byte* is an ASCII byte, prepend *byte* to *ioQueue*.
    8. Return error.
6. If *byte* is an ASCII byte, return a code point whose value is *byte*.
7. If *byte* is 0x80, return code point U+20AC.
8. If *byte* is in the range 0x81 to 0xFE, inclusive, set gb18030 first to *byte* and return continue.
9. Return error.

§ 10.2.2. gb18030 encoder

gb18030's encoder has an associated **is GBK** (initially false).

gb18030's encoder's handler, given *ioQueue* and *code point*, runs these steps:

1. If *code point* is end-of-queue, return finished.
2. If *code point* is an ASCII code point, return a byte whose value is *code point*.
3. If *code point* is U+E5E5, return error with *code point*.

    Note

    Index gb18030 maps 0xA3 0xA0 to U+3000 rather than U+E5E5 for compatibility with deployed content. Therefore it cannot roundtrip.

4. If is GBK is true and *code point* is U+20AC, return byte 0x80.
5. Let *pointer* be the index pointer for *code point* in index gb18030.
6. If *pointer* is non-null, then:
    1. Let *lead* be *pointer* / 190 + 0x81.

   2. Let *trail* be *pointer* % 190.

   3. Let *offset* be 0x40 if *trail* is less than 0x3F, otherwise 0x41.

   4. Return two bytes whose values are *lead* and *trail* + *offset*.

7. If is GBK is true, return error with *code point*.

8. Set *pointer* to the index gb18030 ranges pointer for *code point*.

9. Let *byte1* be *pointer* / (10 × 126 × 10).

10. Set *pointer* to *pointer* % (10 × 126 × 10).

11. Let *byte2* be *pointer* / (10 × 126).

12. Set *pointer* to *pointer* % (10 × 126).

13. Let *byte3* be *pointer* / 10.

14. Let *byte4* be *pointer* % 10.

15. Return four bytes whose values are *byte1* + 0x81, *byte2* + 0x30, *byte3* + 0x81, *byte4* + 0x30.

§ 11. Legacy multi-byte Chinese (traditional) encodings

§ 11.1. Big5

§ 11.1.1. Big5 decoder

Big5’s decoder has an associated **Big5 lead** (initially 0x00).

Big5’s decoder’s handler, given *ioQueue* and *byte*, runs these steps:

1. If *byte* is end-of-queue and Big5 lead is not 0x00, set Big5 lead to 0x00 and return error.
2. If *byte* is end-of-queue and Big5 lead is 0x00, return finished.
3. If Big5 lead is not 0x00, let *lead* be Big5 lead, let *pointer* be null, set Big5 lead to 0x00, and then:
    1. Let *offset* be 0x40 if *byte* is less than 0x7F, otherwise 0x62.
    2. If *byte* is in the range 0x40 to 0x7E, inclusive, or 0xA1 to 0xFE, inclusive, set *pointer* to (*lead* − 0x81) × 157 + (*byte* − *offset*).
    3. If there is a row in the table below whose first column is *pointer*, return the *two* code points listed in its second column (the third column is irrelevant):

Pointer	Code points	Notes
1133	U+00CA U+0304	Ê̄ (LATIN CAPITAL LETTER E WITH CIRCUMFLEX AND MACRON)
1135	U+00CA U+030C	Ê̌ (LATIN CAPITAL LETTER E WITH CIRCUMFLEX AND CARON)
1164	U+00EA U+0304	ê̄ (LATIN SMALL LETTER E WITH CIRCUMFLEX AND MACRON)
1166	U+00EA U+030C	ê̌ (LATIN SMALL LETTER E WITH CIRCUMFLEX AND CARON)

    Note

    Since indexes are limited to single code points this table is used for these pointers.

    4. Let *code point* be null if *pointer* is null, otherwise the index code point for *pointer* in index Big5.
    5. If *code point* is non-null, return a code point whose value is *code point*.
    6. If *byte* is an ASCII byte, prepend *byte* to *ioQueue*.
    7. Return error.
4. If *byte* is an ASCII byte, return a code point whose value is *byte*.
5. If *byte* is in the range 0x81 to 0xFE, inclusive, set Big5 lead to *byte* and return continue.
6. Return error.

§ 11.1.2. Big5 encoder

Big5's encoder's handler, given *ioQueue* and *code point*, runs these steps:

1. If *code point* is end-of-queue, return finished.
2. If *code point* is an ASCII code point, return a byte whose value is *code point*.
3. Let *pointer* be the index Big5 pointer for *code point*.
4. If *pointer* is null, return error with *code point*.
5. Let *lead* be *pointer* / 157 + 0x81.
6. Let *trail* be *pointer* % 157.
7. Let *offset* be 0x40 if *trail* is less than 0x3F, otherwise 0x62.
8. Return two bytes whose values are *lead* and *trail* + *offset*.

§ 12. Legacy multi-byte Japanese encodings

§ 12.1. EUC-JP

§ 12.1.1. EUC-JP decoder

EUC-JP's decoder has an associated **EUC-JP jis0212** (initially false) and **EUC-JP lead** (initially 0x00).

EUC-JP's decoder's handler, given *ioQueue* and *byte*, runs these steps:

1. If *byte* is end-of-queue and EUC-JP lead is not 0x00, set EUC-JP lead to 0x00, and return error.
2. If *byte* is end-of-queue and EUC-JP lead is 0x00, return finished.
3. If EUC-JP lead is 0x8E and *byte* is in the range 0xA1 to 0xDF, inclusive, set EUC-JP lead to 0x00 and return a code point whose value is 0xFF61 − 0xA1 + *byte*.
4. If EUC-JP lead is 0x8F and *byte* is in the range 0xA1 to 0xFE, inclusive, set EUC-JP jis0212 to true, set EUC-JP lead to *byte*, and return continue.
5. If EUC-JP lead is not 0x00, let *lead* be EUC-JP lead, set EUC-JP lead to 0x00, and then:
    1. Let *code point* be null.
    2. If *lead* and *byte* are both in the range 0xA1 to 0xFE, inclusive, then set *code point* to the index code point for (*lead* − 0xA1) × 94 + *byte* − 0xA1 in index jis0208 if EUC-JP jis0212 is false and in index jis0212 otherwise.
    3. Set EUC-JP jis0212 to false.
    4. If *code point* is non-null, return a code point whose value is *code point*.
    5. If *byte* is an ASCII byte, prepend *byte* to *ioQueue*.
    6. Return error.
6. If *byte* is an ASCII byte, return a code point whose value is *byte*.
7. If *byte* is 0x8E, 0x8F, or in the range 0xA1 to 0xFE, inclusive, set EUC-JP lead to *byte* and return continue.
8. Return error.

§ 12.1.2. EUC-JP encoder

EUC-JP's encoder's handler, given *ioQueue* and *code point*, runs these steps:

1. If *code point* is end-of-queue, return finished.
2. If *code point* is an ASCII code point, return a byte whose value is *code point*.
3. If *code point* is U+00A5, return byte 0x5C.
4. If *code point* is U+203E, return byte 0x7E.

5. If *code point* is in the range U+FF61 to U+FF9F, inclusive, return two bytes whose values are 0x8E and *code point* − 0xFF61 + 0xA1.

6. If *code point* is U+2212, set it to U+FF0D.

7. Let *pointer* be the index pointer for *code point* in index jis0208.

   Note

   If pointer *is non-null, it is less than 8836 due to the nature of index jis0208 and the index pointer operation.*

8. If *pointer* is null, return error with *code point*.

9. Let *lead* be *pointer* / 94 + 0xA1.

10. Let *trail* be *pointer* % 94 + 0xA1.

11. Return two bytes whose values are *lead* and *trail*.

§ 12.2. ISO-2022-JP

§ 12.2.1. ISO-2022-JP decoder

ISO-2022-JP's decoder has an associated **ISO-2022-JP decoder state** (initially ASCII), **ISO-2022-JP decoder output state** (initially ASCII), **ISO-2022-JP lead** (initially 0x00), and **ISO-2022-JP output** (initially false).

ISO-2022-JP's decoder's handler, given *ioQueue* and *byte*, runs these steps, switching on ISO-2022-JP decoder state:

↪ ***ASCII***

Based on *byte*:

↪ **0x1B**

Set ISO-2022-JP decoder state to escape start and return continue.

↪ **0x00 to 0x7F, excluding 0x0E, 0x0F, and 0x1B**

Set ISO-2022-JP output to false and return a code point whose value is *byte*.

↪ **end-of-queue**

Return finished.

↪ **Otherwise**

Set ISO-2022-JP output to false and return error.

↪ ***Roman***

Based on *byte*:

↪ **0x1B**

Set ISO-2022-JP decoder state to escape start and return continue.

↪ **0x5C**

Set ISO-2022-JP output to false and return code point U+00A5.

↪ **0x7E**

Set ISO-2022-JP output to false and return code point U+203E.

↪ **0x00 to 0x7F, excluding 0x0E, 0x0F, 0x1B, 0x5C, and 0x7E**

Set ISO-2022-JP output to false and return a code point whose value is *byte*.

↪ **end-of-queue**

Return finished.

↪ **Otherwise**

Set ISO-2022-JP output to false and return error.

↪ ***katakana***

Based on *byte*:

↪ **0x1B**

Set ISO-2022-JP decoder state to escape start and return continue.

↪ **0x21 to 0x5F**

Set ISO-2022-JP output to false and return a code point whose value is 0xFF61 − 0x21 + *byte*.

↪ **end-of-queue**

Return finished.

↪ **Otherwise**

Set ISO-2022-JP output to false and return error.

↪ ***Lead byte***

Based on *byte*:

↪ **0x1B**

Set ISO-2022-JP decoder state to escape start and return continue.

↪ **0x21 to 0x7E**

Set ISO-2022-JP output to false, ISO-2022-JP lead to *byte*, ISO-2022-JP decoder state to trail byte, and return continue.

↪ **end-of-queue**

Return finished.

↪ **Otherwise**

Set ISO-2022-JP output to false and return error.

↪ ***Trail byte***

Based on *byte*:

↪ **0x1B**

Set ISO-2022-JP decoder state to escape start and return error.

↪ **0x21 to 0x7E**

1. Set the ISO-2022-JP decoder state to lead byte.

2. Let *pointer* be (ISO-2022-JP lead − 0x21) × 94 + *byte* − 0x21.

3. Let *code point* be the index code point for *pointer* in index jis0208.

4. If *code point* is null, return error.

5. Return a code point whose value is *code point*.

↪ **end-of-queue**

Set the ISO-2022-JP decoder state to lead byte, prepend *byte* to *ioQueue*, and return error.

↪ **Otherwise**

Set ISO-2022-JP decoder state to lead byte and return error.

↪ ***Escape start***

1. If *byte* is either 0x24 or 0x28, set ISO-2022-JP lead to *byte*, ISO-2022-JP decoder state to escape, and return continue.

2. Prepend *byte* to *ioQueue*.

3. Set ISO-2022-JP output to false, ISO-2022-JP decoder state to ISO-2022-JP decoder output state, and return error.

↪ ***Escape***

1. Let *lead* be ISO-2022-JP lead and set ISO-2022-JP lead to 0x00.

2. Let *state* be null.

3. If *lead* is 0x28 and *byte* is 0x42, set *state* to ASCII.

4. If *lead* is 0x28 and *byte* is 0x4A, set *state* to Roman.

5. If *lead* is 0x28 and *byte* is 0x49, set *state* to katakana.

6. If *lead* is 0x24 and *byte* is either 0x40 or 0x42, set *state* to lead byte.

7. If *state* is non-null, then:

    1. Set ISO-2022-JP decoder state and ISO-2022-JP decoder output state to *state*.

    2. Let *output* be the value of ISO-2022-JP output.

    3. Set ISO-2022-JP output to true.

    4. Return continue, if *output* is false, and error otherwise.

8. Prepend *lead* and *byte* to *ioQueue*.

9. Set ISO-2022-JP output to false, ISO-2022-JP decoder state to ISO-2022-JP decoder output state and return error.

§ 12.2.2. ISO-2022-JP encoder

Note

The ISO-2022-JP encoder is the only encoder for which the concatenation of multiple outputs can result in an error when run through the corresponding decoder.

¶ Example

Encoding U+00A5 gives 0x1B 0x28 0x4A 0x5C 0x1B 0x28 0x42. Doing that twice, concatenating the results, and then decoding yields U+00A5 U+FFFD U+00A5.

ISO-2022-JP's encoder has an associated **ISO-2022-JP encoder state** which is **ASCII**, **Roman**, or **jis0208** (initially ASCII).

ISO-2022-JP's encoder's handler, given *ioQueue* and *code point*, runs these steps:

1. If *code point* is end-of-queue and ISO-2022-JP encoder state is not ASCII, prepend *code point* to *ioQueue*, set ISO-2022-JP encoder state to ASCII, and return three bytes 0x1B 0x28 0x42.
2. If *code point* is end-of-queue and ISO-2022-JP encoder state is ASCII, return finished.
3. If ISO-2022-JP encoder state is ASCII or Roman, and *code point* is U+000E, U+000F, or U+001B, return error with U+FFFD.

   Note

   This returns U+FFFD rather than code point *to prevent attacks.*

4. If ISO-2022-JP encoder state is ASCII and *code point* is an ASCII code point, return a byte whose value is *code point*.
5. If ISO-2022-JP encoder state is Roman and *code point* is an ASCII code point, excluding U+005C and U+007E, or is U+00A5 or U+203E, then:
   1. If *code point* is an ASCII code point, return a byte whose value is *code point*.
   2. If *code point* is U+00A5, return byte 0x5C.
   3. If *code point* is U+203E, return byte 0x7E.
6. If *code point* is an ASCII code point, and ISO-2022-JP encoder state is not ASCII, prepend *code point* to *ioQueue*, set ISO-2022-JP encoder state to ASCII, and return three bytes 0x1B 0x28 0x42.
7. If *code point* is either U+00A5 or U+203E, and ISO-2022-JP encoder state is not Roman, prepend *code point* to *ioQueue*, set ISO-2022-JP encoder state to Roman, and return three bytes 0x1B 0x28 0x4A.
8. If *code point* is U+2212, set it to U+FF0D.
9. If *code point* is in the range U+FF61 to U+FF9F, inclusive, set it to the index code point for *code point* − 0xFF61 in index ISO-2022-JP katakana.
10. Let *pointer* be the index pointer for *code point* in index jis0208.

    Note

    If pointer *is non-null, it is less than 8836 due to the nature of index jis0208* *and the index pointer operation.*

11. If *pointer* is null, then:

    1. If ISO-2022-JP encoder state is jis0208, then prepend *code point* to *ioQueue*, set ISO-2022-JP encoder state to ASCII, and return three bytes 0x1B 0x28 0x42.

    2. Return error with *code point*.

12. If ISO-2022-JP encoder state is not jis0208, prepend *code point* to *ioQueue*, set ISO-2022-JP encoder state to jis0208, and return three bytes 0x1B 0x24 0x42.

13. Let *lead* be *pointer* / 94 + 0x21.

14. Let *trail* be *pointer* % 94 + 0x21.

15. Return two bytes whose values are *lead* and *trail*.

§ 12.3. Shift_JIS

§ 12.3.1. Shift_JIS decoder

Shift_JIS's decoder has an associated **Shift_JIS lead** (initially 0x00).

Shift_JIS's decoder's handler, given *ioQueue* and *byte*, runs these steps:

1. If *byte* is end-of-queue and Shift_JIS lead is not 0x00, set Shift_JIS lead to 0x00 and return error.

2. If *byte* is end-of-queue and Shift_JIS lead is 0x00, return finished.

3. If Shift_JIS lead is not 0x00, let *lead* be Shift_JIS lead, let *pointer* be null, set Shift_JIS lead to 0x00, and then:

    1. Let *offset* be 0x40 if *byte* is less than 0x7F, otherwise 0x41.

    2. Let *lead offset* be 0x81 if *lead* is less than 0xA0, otherwise 0xC1.

    3. If *byte* is in the range 0x40 to 0x7E, inclusive, or 0x80 to 0xFC, inclusive, set *pointer* to (*lead* − *lead offset*) × 188 + *byte* − *offset*.

    4. If *pointer* is in the range 8836 to 10715, inclusive, return a code point whose value is 0xE000 − 8836 + *pointer*.

       Note

       This is interoperable legacy from Windows known as EUDC.

    5. Let *code point* be null if *pointer* is null, otherwise the index code point for *pointer* in index jis0208.

    6. If *code point* is non-null, return a code point whose value is *code point*.

    7. If *byte* is an ASCII byte, prepend *byte* to *ioQueue*.

    8. Return error.

4. If *byte* is an ASCII byte or 0x80, return a code point whose value is *byte*.

5. If *byte* is in the range 0xA1 to 0xDF, inclusive, return a code point whose value is 0xFF61 − 0xA1 + *byte*.

6. If *byte* is in the range 0x81 to 0x9F, inclusive, or 0xE0 to 0xFC, inclusive, set Shift_JIS lead to *byte* and return continue.

7. Return error.

§ 12.3.2. Shift_JIS encoder

Shift_JIS’s encoder’s handler, given *ioQueue* and *code point*, runs these steps:

1. If *code point* is end-of-queue, return finished.

2. If *code point* is an ASCII code point or U+0080, return a byte whose value is *code point*.

3. If *code point* is U+00A5, return byte 0x5C.

4. If *code point* is U+203E, return byte 0x7E.

5. If *code point* is in the range U+FF61 to U+FF9F, inclusive, return a byte whose value is *code point* − 0xFF61 + 0xA1.

6. If *code point* is U+2212, set it to U+FF0D.

7. Let *pointer* be the index Shift_JIS pointer for *code point*.

8. If *pointer* is null, return error with *code point*.

9. Let *lead* be *pointer* / 188.

10. Let *lead offset* be 0x81 if *lead* is less than 0x1F, otherwise 0xC1.

11. Let *trail* be *pointer* % 188.

12. Let *offset* be 0x40 if *trail* is less than 0x3F, otherwise 0x41.

13. Return two bytes whose values are *lead* + *lead offset* and *trail* + *offset*.

§ 13. Legacy multi-byte Korean encodings

§ 13.1. EUC-KR

§ 13.1.1. EUC-KR decoder

EUC-KR's decoder has an associated **EUC-KR lead** (initially 0x00).

EUC-KR's decoder's handler, given *ioQueue* and *byte*, runs these steps:

1. If *byte* is end-of-queue and EUC-KR lead is not 0x00, set EUC-KR lead to 0x00 and return error.
2. If *byte* is end-of-queue and EUC-KR lead is 0x00, return finished.
3. If EUC-KR lead is not 0x00, let *lead* be EUC-KR lead, let *pointer* be null, set EUC-KR lead to 0x00, and then:
    1. If *byte* is in the range 0x41 to 0xFE, inclusive, set *pointer* to (*lead* − 0x81) × 190 + (*byte* − 0x41).
    2. Let *code point* be null if *pointer* is null, otherwise the index code point for *pointer* in index EUC-KR.
    3. If *code point* is non-null, return a code point whose value is *code point*.
    4. If *byte* is an ASCII byte, prepend *byte* to *ioQueue*.
    5. Return error.
4. If *byte* is an ASCII byte, return a code point whose value is *byte*.
5. If *byte* is in the range 0x81 to 0xFE, inclusive, set EUC-KR lead to *byte* and return continue.
6. Return error.

§ 13.1.2. EUC-KR encoder

EUC-KR's encoder's handler, given *ioQueue* and *code point*, runs these steps:

1. If *code point* is end-of-queue, return finished.
2. If *code point* is an ASCII code point, return a byte whose value is *code point*.
3. Let *pointer* be the index pointer for *code point* in index EUC-KR.
4. If *pointer* is null, return error with *code point*.
5. Let *lead* be *pointer* / 190 + 0x81.
6. Let *trail* be *pointer* % 190 + 0x41.
7. Return two bytes whose values are *lead* and *trail*.

§ 14. Legacy miscellaneous encodings

§ 14.1. replacement

Note

The replacement encoding exists to prevent certain attacks that abuse a mismatch between encodings supported on the server and the client.

§ 14.1.1. replacement decoder

replacement's decoder has an associated **replacement error returned** (initially false).

replacement's decoder's handler, given *ioQueue* and *byte*, runs these steps:

1. If *byte* is end-of-queue, return finished.
2. If replacement error returned is false, set replacement error returned to true and return error.
3. Return finished.

§ 14.2. Common infrastructure for UTF-16BE/LE

UTF-16BE/LE is UTF-16BE or UTF-16LE.

§ 14.2.1. shared UTF-16 decoder

Note

A byte order mark has priority over a label as it has been found to be more accurate in deployed content. Therefore it is not part of the shared UTF-16 decoder algorithm but rather the decode algorithm.

shared UTF-16 decoder has an associated **UTF-16 lead byte** and **UTF-16 lead surrogate** (both initially null), and **is UTF-16BE decoder** (initially false).

shared UTF-16 decoder's handler, given *ioQueue* and *byte*, runs these steps:

1. If *byte* is end-of-queue and either UTF-16 lead byte or UTF-16 lead surrogate is non-null, set UTF-16 lead byte and UTF-16 lead surrogate to null, and return error.
2. If *byte* is end-of-queue and UTF-16 lead byte and UTF-16 lead surrogate are null, return finished.
3. If UTF-16 lead byte is null, set UTF-16 lead byte to *byte* and return continue.
4. Let *code unit* be the result of:

↪ **is UTF-16BE decoder is true**

(UTF-16 lead byte << 8) + *byte*.

↪ **is UTF-16BE decoder is false**

(*byte* << 8) + UTF-16 lead byte.

Then set UTF-16 lead byte to null.

5. If UTF-16 lead surrogate is non-null, let *lead surrogate* be UTF-16 lead surrogate, set UTF-16 lead surrogate to null, and then:
    1. If *code unit* is in the range U+DC00 to U+DFFF, inclusive, return a code point whose value is 0x10000 + ((*lead surrogate* − 0xD800) << 10) + (*code unit* − 0xDC00).
    2. Let *byte1* be *code unit* >> 8.
    3. Let *byte2* be *code unit* & 0x00FF.
    4. Let *bytes* be two bytes whose values are *byte1* and *byte2*, if is UTF-16BE decoder is true, and *byte2* and *byte1* otherwise.
    5. Prepend the *bytes* to *ioQueue* and return error.
6. If *code unit* is in the range U+D800 to U+DBFF, inclusive, set UTF-16 lead surrogate to *code unit* and return continue.
7. If *code unit* is in the range U+DC00 to U+DFFF, inclusive, return error.
8. Return code point *code unit*.

§ 14.3. UTF-16BE

§ 14.3.1. UTF-16BE decoder

UTF-16BE's decoder is shared UTF-16 decoder with its is UTF-16BE decoder set to true.

§ 14.4. UTF-16LE

Note

"`utf-16`" *is a label for UTF-16LE to deal with deployed content.*

§ 14.4.1. UTF-16LE decoder

UTF-16LE's decoder is shared UTF-16 decoder.

§ 14.5. x-user-defined

Note

While technically this is a single-byte encoding, it is defined separately as it can be implemented algorithmically.

§ 14.5.1. x-user-defined decoder

x-user-defined's decoder's handler, given *ioQueue* and *byte*, runs these steps:

1. If *byte* is end-of-queue, return finished.
2. If *byte* is an ASCII byte, return a code point whose value is *byte*.
3. Return a code point whose value is 0xF780 + *byte* − 0x80.

§ 14.5.2. x-user-defined encoder

x-user-defined's encoder's handler, given *ioQueue* and *code point*, runs these steps:

1. If *code point* is end-of-queue, return finished.
2. If *code point* is an ASCII code point, return a byte whose value is *code point*.
3. If *code point* is in the range U+F780 to U+F7FF, inclusive, return a byte whose value is *code point* − 0xF780 + 0x80.
4. Return error with *code point*.

§ 15. Browser UI

Browsers are encouraged to not enable overriding the encoding of a resource. If such a feature is nonetheless present, browsers should not offer UTF-16BE/LE as an option, due to the aforementioned security issues. Browsers should also disable this feature if the resource was decoded using UTF-16BE/LE.

§ Implementation considerations

Instead of supporting I/O queues with arbitrary prepend, the decoders for encodings in this standard could be implemented with:

1. The ability to unread the current byte.
2. A single-byte buffer for gb18030 (an ASCII byte) and ISO-2022-JP (0x24 or 0x28).

¶ Example

For gb18030 when hitting a bogus byte while gb18030 third is not 0x00, gb18030 second could be moved into the single-byte buffer to be returned next, and gb18030 third would be the new gb18030 first, checked for not being 0x00 after the single-byte buffer was returned and emptied. This is possible as the range for the first and third byte in gb18030 is identical.

The ISO-2022-JP encoder needs ISO-2022-JP encoder state as additional state, but other than that, none of the encoders for encodings in this standard require additional state or buffers.

§ Acknowledgments

There have been a lot of people that have helped make encodings more interoperable over the years and thereby furthered the goals of this standard. Likewise many people have helped making this standard what it is today.

With that, many thanks to Adam Rice, Alan Chaney, Alexander Shtuchkin, Allen Wirfs-Brock, Andreu Botella, Aneesh Agrawal, Arkadiusz Michalski, Asmus Freytag, Ben Noordhuis, Bnaya Peretz, Boris Zbarsky, Bruno Haible, Cameron McCormack, Charles McCathieNeville, Christopher Foo, CodifierNL, David Carlisle, Domenic Denicola, Dominique Hazaël-Massieux, Doug Ewell, Erik van der Poel, 譚永鋒 (Frank Yung-Fong Tang), Glenn Maynard, Gordon P. Hemsley, Henri Sivonen, Ian Hickson, J. King, James Graham, Jeffrey Yasskin, John Tamplin, Joshua Bell, 村井純 (Jun Murai), 신정식 (Jungshik Shin), Jxck, 강 성훈 (Kang Seonghoon), 川幡太一 (Kawabata Taichi), Ken Lunde, Ken Whistler, Kenneth Russell, 田村健人 (Kent Tamura), Leif Halvard Silli, Luke Wagner, Maciej Hirsz, Makoto Kato, Mark Callow, Mark Crispin, Mark Davis, Martin Dürst, Masatoshi Kimura, Mattias Buelens, Ms2ger, Nigel Megitt, Nigel Tao, Norbert Lindenberg, Øistein E. Andersen, Peter Krefting, Philip Jägenstedt, Philip Taylor, Richard Ishida, Robbert Broersma, Robert Mustacchi, Ryan Dahl, Sam Sneddon, Shawn Steele, Simon Montagu, Simon Pieters, Simon Sapin, Stephen Checkoway, 寺田健 (Takeshi Terada), Vyacheslav Matva, Wolf Lammen, and 成瀬ゆい (Yui Naruse) for being awesome.

This standard is written by Anne van Kesteren (Mozilla, annevk@annevk.nl). The API chapter was initially written by Joshua Bell (Google).

§ Intellectual property rights

This is the Living Standard. Those interested in the patent-review version should view the Living Standard Review Draft.

§ Index

§ Terms defined by this specification

§ Terms defined by reference

- [HTML] defines the following terms:
  - event loop
  - in parallel
- [INFRA] defines the following terms:
  - append
  - ascii byte
  - ascii case-insensitive
  - ascii code point
  - ascii lowercase
  - ascii whitespace
  - break
  - byte
  - byte sequence
  - code point
  - code unit
  - contain
  - continue
  - convert
  - empty
  - for each
  - insert
  - item
  - length
  - list
  - prepend
  - queue
  - remove
  - scalar value
  - scalar value string
  - size
  - starts with
  - string
  - surrogate
  - the range
  - value
- [STREAMS] defines the following terms:
  - GenericTransformStream
  - ReadableStream
  - TransformStream
  - chunk
  - creating
  - enqueue
  - flushalgorithm
  - pipeThrough(transform)
  - readable
  - readable stream
  - transform
  - transformalgorithm
  - writable
  - writable stream
- [WEBIDL] defines the following terms:
  - AllowShared
  - ArrayBuffer
  - BufferSource

- DOMString
- Exposed
- NewObject
- RangeError
- TypeError
- USVString
- Uint32Array
- Uint8Array
- boolean
- converted to an idl value
- get a copy of the buffer source
- get a reference to the buffer source
- this
- throw
- unsigned long long

§ References

§ Normative References

[INFRA]

Anne van Kesteren; Domenic Denicola. Infra Standard. Living Standard. URL: https://infra.spec.whatwg.org/

[STREAMS]

Adam Rice; Domenic Denicola; 吉野剛史 (Takeshi Yoshino). Streams Standard. Living Standard. URL: https://streams.spec.whatwg.org/

[UNICODE]

The Unicode Standard. URL: https://www.unicode.org/versions/latest/

[WEBIDL]

Boris Zbarsky. Web IDL. URL: https://heycam.github.io/webidl/

§ Informative References

[HTML]

Anne van Kesteren; et al. HTML Standard. Living Standard. URL: https://html.spec.whatwg.org/multipage/

[URL]

Anne van Kesteren. URL Standard. Living Standard. URL: https://url.spec.whatwg.org/

[XML]

Tim Bray; et al. Extensible Markup Language (XML) 1.0 (Fifth Edition). 26 November 2008. REC. URL: https://www.w3.org/TR/xml/

§ IDL Index

```
interface mixin TextDecoderCommon {
  readonly attribute DOMString encoding;
  readonly attribute boolean fatal;
  readonly attribute boolean ignoreBOM;
};

dictionary TextDecoderOptions {
  boolean fatal = false;
  boolean ignoreBOM = false;
};

dictionary TextDecodeOptions {
  boolean stream = false;
};

[Exposed=(Window,Worker)]
interface TextDecoder {
  constructor(optional DOMString label = "utf-8", optional
TextDecoderOptions options = {});

  USVString decode(optional [AllowShared] BufferSource input,
optional TextDecodeOptions options = {});
};
TextDecoder includes TextDecoderCommon;

interface mixin TextEncoderCommon {
  readonly attribute DOMString encoding;
};

dictionary TextEncoderEncodeIntoResult {
  unsigned long long read;
  unsigned long long written;
};

[Exposed=(Window,Worker)]
interface TextEncoder {
  constructor();

  [NewObject] Uint8Array encode(optional USVString input =
"");
  TextEncoderEncodeIntoResult encodeInto(USVString source,
[AllowShared] Uint8Array destination);
};
TextEncoder includes TextEncoderCommon;

[Exposed=(Window,Worker)]
interface TextDecoderStream {
  constructor(optional DOMString label = "utf-8", optional
TextDecoderOptions options = {});
};
TextDecoderStream includes TextDecoderCommon;
TextDecoderStream includes GenericTransformStream;
```

```
[Exposed=(Window,Worker)]
interface TextEncoderStream {
  constructor();
};
TextEncoderStream includes TextEncoderCommon;
TextEncoderStream includes GenericTransformStream;
```