L2/12-176

Encoding

Living Specification — Last Updated 4 May 2012

This Version:

http://dvcs.w3.org/hg/encoding/raw-file/tip/Overview.html

Participate:

Send feedback to whatwg@whatwg.org (archives) or file a bug (open bugs)
IRC: #whatwg on Freenode

Version History:

http://dvcs.w3.org/hg/encoding/shortlog

Editor:

Anne van Kesteren (Opera Software ASA) <annevk@annevk.nl>

Disclaimer

Table of Contents

1 Preface

While encodings for the web platform have been defined to some extent, implementations 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 attempts to fill those gaps so that new implementations do not have to reverse engineer encoding implementations of the market leaders and existing implementations can become more interoperable.

> ***Note: This specification is primarily intended for dealing with legacy content, it requires new content and formats to use the utf-8 encoding exclusively.***

2 Conformance

All diagrams, examples, and notes in this specification are non-normative, as are all sections explicitly marked non-normative. Everything else in this specification is normative.

The key words "MUST", "MUST NOT", "REQUIRED", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in the normative parts of this document are to be interpreted as described in RFC2119. For readability, these words do not appear in all uppercase letters in this specification. [RFC2119]

Conformance requirements phrased as algorithms or specific steps may be implemented in any manner, so long as the end result is equivalent. (In particular, the algorithms defined in this specification are intended to be easy to follow, and not intended to be performant.)

User agents may impose implementation-specific limits on otherwise unconstrained inputs, e.g. to prevent denial of service attacks, to guard against running out of memory, or to work around platform-specific limitations.

3 Terminology

Hexadecimal numbers are prefixed with "0x".

In equations, all numbers are integers, addition is represented by "+", subtraction by "−", multiplication by "×", division by "/", calculating the remainder of a division (also known as modulo) by "%", exponentiation by "b^n", arithmetic left shifts by "<<", arithmetic right shifts by ">>", bitwise AND by "&", and bitwise OR by "|".

A **byte** is referenced as a double-digit hexadecimal number in the range 0x00 to 0xFF.

Note: Web platform bytes consist of exactly eight bits.

A **code point** is a Unicode code point and is referenced as a four-to-six digit hexadecimal number, typically prefixed with "U+". In equations and indexes it is prefixed with "0x". [UNICODE]

The **space characters**, for the purposes of this specification, are U+0020 SPACE, U+0009 CHARACTER TABULATION (tab), U+000A LINE FEED (LF), U+000C FORM FEED (FF), and U+000D CARRIAGE RETURN (CR).

Comparing two strings in an **ASCII case-insensitive** manner means comparing them exactly, code point for code point, except that the characters in the range U+0041 to U+005A (i.e. LATIN CAPITAL LETTER A to LATIN CAPITAL LETTER Z) and the corresponding characters in the range U+0061 to U+007A (i.e. LATIN SMALL LETTER A to LATIN SMALL LETTER Z) are considered to also match.

4 Encodings

An **encoding** defines a mapping from a code point to one or more bytes (and vice versa). Each encoding has a **name**, and one or more **labels**.

Each encoding also has a **decoder** and **encoder** algorithm.

A decoder algorithm takes a stream of bytes and emits a stream of code points. The **byte pointer** is initially zero, pointing to the first byte in the stream. It cannot be negative. It can be increased and decreased to point to other bytes in the stream. The **EOF byte** is a conceptual byte representing the end of the stream. The byte pointer cannot point beyond the EOF byte. The **EOF code point** is a conceptual code point that is emitted once the stream of bytes is handled in its entirety. A **decoder error** indicates an error in the stream of bytes. Unless stated otherwise emitting a decoder error must emit code point U+FFFD. A decoder can end by emitting a code point or decoder error, or the word continue. Unless the EOF code point is emitted, the decoder algorithm must be invoked again.

> ***Note: An XML processor would halt upon the first decoder error emitted.***

An encoder algorithm takes a stream of code points and emits a stream of bytes. It will fail when a code point is passed for which it does not have a corresponding byte or byte sequence. Analogously to a decoder, it has a **code point pointer** and **encoder error**. Unless stated otherwise emitting an encoder error terminates the encoder. Again analogously, as long as the EOF byte is not emitted, the encoder algorithm must be invoked for each byte or sequence of bytes emitted.

> ***Note: HTML forms and URLs require non-terminating encoders and have therefore special handling whenever an encoder error is reached. Instead of terminating the encoder a sequence of one or more code points in the range U+0000 to U+007F (representing the code point that caused the encoder error to be emitted) are inserted into the stream of code points at code point pointer after encoder error is emitted.***

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

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

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

> ***Note: This algorithm is different from the one defined in section 1.4 of Unicode Technical Standard #22 as that algorithm is incompatible with legacy content.***

Name	Labels
The Encoding	
utf-8	"`unicode-1-1-utf-8`"
	"`utf-8`"
	"`utf8`"
Legacy single-byte encodings	
ibm864	"`cp864`"
	"`csibm864`"
	"`ibm-864`"
	"`ibm864`"
ibm866	"`866`"
	"`cp866`"
	"`csibm866`"
	"`ibm866`"
iso-8859-2	"`csisolatin2`"
	"`iso-8859-2`"
	"`iso-ir-101`"
	"`iso8859-2`"
	"`iso88592`"
	"`iso_8859-2`"

Name	Labels
	"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"
	"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"

Name	Labels
	"iso88597"
	"iso_8859-7"
	"iso_8859-7:1987"
	"sun_eu_greek"
iso-8859-8	"csiso88598e"
	"csiso88598i"
	"csisolatinhebrew"
	"hebrew"
	"iso-8859-8"
	"iso-8859-8-e"
	"iso-8859-8-i"
	"iso-ir-138"
	"iso8859-8"
	"iso88598"
	"iso_8859-8"
	"iso_8859-8:1988"
	"logical"
	"visual"
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-u"
macintosh	"csmacintosh"
	"mac"
	"macintosh"
	"x-mac-roman"
windows-874	"dos-874"
	"iso-8859-11"
	"iso8859-11"

Name	Labels
	"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"
	"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"

Name	Labels
	"windows-1258"
	"x-cp1258"
x-mac-cyrillic	"x-mac-cyrillic"
	"x-mac-ukrainian"
Legacy multi-byte Chinese (simplified) encodings	
gbk	"chinese"
	"csgb2312"
	"csiso58gb231280"
	"gb2312"
	"gb_2312"
	"gb_2312-80"
	"gbk"
	"iso-ir-58"
	"x-gbk"
gb18030	"gb18030"
hz-gb-2312	"hz-gb-2312"
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"
	"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"
iso-2022-kr	"csiso2022kr"
	"iso-2022-kr"
Legacy utf-16 encodings	
utf-16	"utf-16"
	"utf-16le"

Name	Labels
utf-16be	"utf-16be"

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

5 Indexes

Most legacy encodings make use of an **index**. An index is an ordered list of pointers and corresponding code points. Within an index pointers are unique and code points can be duplicated.

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.

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*.

These are the indexes defined by this specification, excluding index single-byte:

Index		Notes
index big5	index-big5.txt	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	This matches the KS X 1001 standard and the Unified Hangul Code, more commonly known together as Windows Codepage 949.
index gbk	index-gbk.txt	This matches the GB18030 standard for code points encoded as two bytes.
index gb18030	index-gb18030.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 standard for code points encoded as four bytes. See also index gb18030 code point and index gb18030 pointer below.
index jis0208	index-jis0208.txt	This is the JIS X 0208 standard including formerly proprietary extensions from IBM and NEC.
index jis0212	index-jis0212.txt	This is the JIS X 0212 standard.

The **index gb18030 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. Let *offset* be the last pointer in index gb18030 that is equal to or less than *pointer* and let *code point offset* be its corresponding code point.
3. Return a code point whose value is *code point offset* + *pointer* − *offset*.

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

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

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

6 Decode and encode

Note: The algorithms decode, utf-8 decode, and encode are intended for usage by other specifications. utf-8 decode is to be used by new formats. The get an encoding algorithm can be used first to turn a label into an encoding.

To **decode** a byte stream *stream* using fallback encoding *encoding*, run these steps:

1. Let *offset* be 0.
2. For each of the rows in the following table, starting with the first one and going down, if the first bytes of *stream* match all the bytes given in the first column (ergo *stream* contains at least two or three bytes), then set *encoding* to the encoding given in the cell in the second column of that row, and set *offset* to the offset given in the cell in the third column of that row.

Byte order mark	Encoding	Offset
0xFF 0xFE	utf-16	2
0xFE 0xFF	utf-16be	2
0xEF 0xBB 0xBF	utf-8	3

Note: For compatibility with deployed content, the byte order mark (also known as BOM) is considered more authoritative than anything else.

3. Return the result of running *encoding*'s decoder with byte pointer set to *offset*, on *stream*.

To **utf-8 decode** a byte stream *stream*, run these steps:

1. Let *offset* be 0.
2. If *stream* contains at least three bytes and its first three bytes match 0xEF 0xBB 0xBF, set *offset* to 3.
3. Return the result of running the utf-8 decoder with byte pointer set to *offset*, on *stream*.

To **encode** a code point stream *stream* using encoding *encoding*, return the result of running *encoding*'s encoder on *stream*.

7 The encoding

New content and formats must exclusively use the utf-8 encoding.

7.1 utf-8

The **utf-8 code point**, **utf-8 bytes seen**, **utf-8 bytes needed**, and **utf-8 lower boundary** concepts are all initially 0.

The **utf-8 decoder** (decoder for utf-8) is:

1. Let *byte* be byte pointer.
2. If *byte* is the EOF byte and utf-8 bytes needed is not 0, set utf-8 bytes needed to 0 and emit a decoder error.
3. If *byte* is the EOF byte, emit the EOF code point.
4. Increase the byte pointer.
5. If utf-8 bytes needed is 0, based on *byte*:

   ↪ **0x00 to 0x7F**
   : Emit a code point whose value is *byte*.

   ↪ **0xC2 to 0xDF**
   : Set utf-8 bytes needed to 1, utf-8 lower boundary to 0x80, and utf-8 code point to *byte* − 0xC0.

   ↪ **0xE0 to 0xEF**
   : Set utf-8 bytes needed to 2, utf-8 lower boundary to 0x800, and utf-8 code point to *byte* − 0xE0.

   ↪ **0xF0 to 0xF4**
   : Set utf-8 bytes needed to 3, utf-8 lower boundary to 0x10000, and utf-8 code point to *byte* − 0xF0.

   ↪ **Otherwise**
   : Emit a decoder error.

   Then (*byte* is in the range 0xC2 to 0xF4) set utf-8 code point to utf-8 code point × $64^{\text{utf-8 bytes needed}}$ and continue.
6. If *byte* is not in the range 0x80 to 0xBF, run these substeps:
   1. Set utf-8 code point, utf-8 bytes needed, utf-8 bytes seen, and utf-8 lower boundary to 0.
   2. Decrease the byte pointer by one.
   3. Emit a decoder error.
7. Increase utf-8 bytes seen by one and set utf-8 code point to utf-8 code point + (*byte* − 0x80) × $64^{\text{utf-8 bytes needed} - \text{utf-8 bytes seen}}$
8. If utf-8 bytes seen is not equal to utf-8 bytes needed, continue.
9. Let *code point* be utf-8 code point and *lower boundary* be utf-8 lower boundary.
10. Set utf-8 code point, utf-8 bytes needed, utf-8 bytes seen, and utf-8 lower boundary to 0.
11. If *code point* is in the range *lower boundary* to 0x10FFFF and is not in the range 0xD800 to 0xDFFF, emit a code point whose value is *code point*.
12. Emit a decoder error.

The **utf-8 encoder** (encoder for utf-8) is:

1. Let *code point* be code point pointer.
2. If *code point* is in the range 0xD800 to 0xDFFF, emit an encoder error.
3. If *code point* is the EOF code point, emit the EOF byte.
4. Increase the code point pointer by one.
5. If *code point* is in the range U+0000 to U+007F, emit a byte whose value is *code point*.

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

   ↪ **U+0080 to U+07FF**
   1 and 0xC0

   ↪ **U+0800 to U+FFFF**
   2 and 0xE0

   ↪ **U+10000 to U+10FFFF**
   3 and 0xF0

7. Let *bytes* be a list of bytes whose first byte is *code point* / 64^{count} + *offset*.

8. Run these substeps while *count* is greater than 0:

   1. Set *temp* to *code point* / $64^{count - 1}$.

   2. Append to *bytes* 0x80 + (*temp* % 64).

   3. Decrease *count* by one.

9. Emit bytes *bytes*, in list order.

8 Legacy single-byte encodings

An encoding where each byte is either a single code point or nothing, is a **single-byte encoding**. All single-byte encodings use the same decoder and encoder, but use a different index. **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.

Name	Index
ibm864	index-ibm864.txt
ibm866	index-ibm866.txt
iso-8859-2	index-iso-8859-2.txt
iso-8859-3	index-iso-8859-3.txt
iso-8859-4	index-iso-8859-4.txt
iso-8859-5	index-iso-8859-5.txt
iso-8859-6	index-iso-8859-6.txt
iso-8859-7	index-iso-8859-7.txt
iso-8859-8	index-iso-8859-8.txt
iso-8859-10	index-iso-8859-10.txt
iso-8859-13	index-iso-8859-13.txt
iso-8859-14	index-iso-8859-14.txt
iso-8859-15	index-iso-8859-15.txt
iso-8859-16	index-iso-8859-16.txt
koi8-r	index-koi8-r.txt
koi8-u	index-koi8-u.txt
macintosh	index-macintosh.txt
windows-874	index-windows-874.txt
windows-1250	index-windows-1250.txt
windows-1251	index-windows-1251.txt
windows-1252	index-windows-1252.txt
windows-1253	index-windows-1253.txt
windows-1254	index-windows-1254.txt
windows-1255	index-windows-1255.txt
windows-1256	index-windows-1256.txt
windows-1257	index-windows-1257.txt
windows-1258	index-windows-1258.txt
x-mac-cyrillic	index-x-mac-cyrillic.txt

The **single-byte decoder** (decoder for single-byte encodings) is:

1. Let *byte* be byte pointer.
2. If *byte* is the EOF byte, emit the EOF code point.
3. Increase the byte pointer by one.
4. If *byte* is in the range 0x00 to 0x7F, emit a code point whose value is *byte*.
5. Let *code point* be the index code point for *byte* − 0x80 in index single-byte.
6. If *code point* is null, emit a decoder error.
7. Emit a code point whose value is *code point*.

The **single-byte encoder** (encoder for single-byte encodings) is:

1. Let *code point* be code point pointer.
2. If *code point* is the EOF code point, emit the EOF byte.
3. Increase the code point pointer by one.
4. If *code point* is in the range U+0000 to U+007F, emit a byte whose value is *code point*.
5. Let *pointer* be the index pointer for *code point* in index single-byte.

6. If *pointer* is null, emit an encoder error.
7. Emit a byte whose value is *pointer* + 0x80.

9 Legacy multi-byte Chinese (simplified) encodings

9.1 gbk

The **gb18030 flag** flag is initially unset. It can only be set by the gb18030 decoder and gb18030 encoder.

The **gbk first**, **gbk second**, and **gbk third**, are all initially 0x00.

The **gbk decoder** (decoder for gbk) is:

1. Let *byte* be byte pointer.
2. If *byte* is the EOF byte and gbk first, gbk second, and gbk third are 0x00, emit the EOF code point.
3. If *byte* is the EOF byte, and gbk first, gbk second, or gbk third is not 0x00, set gbk first, gbk second, and gbk third to 0x00, and emit a decoder error.
4. Increase the byte pointer.
5. If gbk third is not 0x00, run these substeps:
    1. Let *code point* be null.
    2. If *byte* is in the range 0x30 to 0x39, set *code point* to the index gb18030 code point for (((gbk first − 0x81) × 10 + gbk second − 0x30) × 126 + gbk third − 0x81) × 10 + *byte* − 0x30.
    3. Set gbk first, gbk second, and gbk third to 0x00.
    4. If *code point* is null, decrease the byte pointer by three and emit a decoder error.
    5. Emit a code point whose value is *code point*.
6. If gbk second is not 0x00, run these substeps:
    1. If *byte* is in the range 0x81 to 0xFE, set gbk third to *byte* and continue.
    2. Decrease the byte pointer by two, set gbk first and gbk second to 0x00, and emit a decoder error.
7. If gbk first is not 0x00, run these substeps:
    1. If *byte* is in the range 0x30 to 0x39 and the gb18030 flag is set, set gbk second to *byte* and continue.
    2. Let *lead* be gbk first, let *pointer* be null, and set gbk first to 0x00.
    3. Let *offset* be 0x40 if *byte* is less than 0x7F, or 0x41 otherwise.
    4. If *byte* is in the range 0x40 to 0x7E or 0x80 to 0xFE, set *pointer* to (*lead* − 0x81) × 190 + (*byte* − *offset*).
    5. Let *code point* be null if *pointer* is null, or the index code point for *pointer* in index gbk otherwise.
    6. If *pointer* is null, decrease the byte pointer by one.
    7. If *code point* is null, emit a decoder error.
    8. Emit a code point whose value is *code point*.
8. If *byte* is in the range 0x00 to 0x7F, emit a code point whose value is *byte*.
9. If *byte* is 0x80, emit code point U+20AC.
10. If *byte* is in the range 0x81 to 0xFE, set gbk first to *byte* and continue.
11. Emit a decoder error.

The **gbk encoder** (encoder for gbk) is:

1. Let *code point* be code point pointer.
2. If *code point* is the EOF code point, emit the EOF byte.
3. Increase the code point pointer by one.
4. If *code point* is in the range U+0000 to U+007F, emit a byte whose value is *code point*.
5. Let *pointer* be the index pointer for *code point* in index gbk.

6. If *pointer* is not null, run these substeps:
    1. Let *lead* be *pointer* / 190 + 0x81.
    2. Let *trail* be *pointer* % 190.
    3. Let *offset* be 0x40 if *trail* is less than 0x3F, or 0x41 otherwise.
    4. Emit two bytes whose values are *lead* and *trail* + *offset*.
7. If *pointer* is null and the gb18030 flag is unset, emit an encoder error.
8. Set *pointer* to the index gb18030 pointer for *code point*.
9. Let *byte1* be *pointer* / 10 / 126 / 10.
10. Set *pointer* to *pointer* − *byte1* × 10 × 126 × 10.
11. Let *byte2* be *pointer* / 10 / 126.
12. Set *pointer* to *pointer* − *byte2* × 10 × 126.
13. Let *byte3* be *pointer* / 10.
14. Let *byte4* be *pointer* − *byte3* × 10.
15. Emit four bytes whose values are *byte1* + 0x81, *byte2* + 0x30, *byte3* + 0x81, *byte4* + 0x30.

9.2 gb18030

The **gb18030 decoder** (decoder for gb18030) is the gbk decoder with the gb18030 flag set.

The **gb18030 encoder** (encoder for gb18030) is the gbk encoder with the gb18030 flag set.

9.3 hz-gb-2312

The **hz-gb-2312 flag** is initially unset. The **hz-gb-2312 lead** is initially 0x00.

The **hz-gb-2312 decoder** (decoder for hz-gb-2312) is:

1. Let *byte* be byte pointer.
2. If *byte* is the EOF byte and hz-gb-2312 lead is 0x00, emit the EOF code point.
3. If *byte* is the EOF byte and hz-gb-2312 lead is not 0x00, set hz-gb-2312 lead to 0x00 and emit a decoder error.
4. Increase the byte pointer.
5. If hz-gb-2312 lead is 0x7E, set hz-gb-2312 lead to 0x00, and based on *byte*:

    ↪ **0x7B**
    : Set the hz-gb-2312 flag and continue.

    ↪ **0x7D**
    : Unset the hz-gb-2312 flag and continue.

    ↪ **0x7E**
    : Emit code point U+007E.

    ↪ **0x0A**
    : Continue.

    ↪ **Otherwise**
    : Decrease the byte pointer by one and emit a decoder error.

6. If hz-gb-2312 lead is not 0x00, let *lead* be hz-gb-2312 lead, set hz-gb-2312 lead to 0x00, and then run these substeps:
    1. If *byte* is in the range 0x21 to 0x7E, let *code point* be the index code point for (*lead* − 1) × 190 + (*byte* + 0x3F) in index gbk.

    2. If *byte* is 0x0A, unset the hz-gb-2312 flag.
    3. If *code point* is null, emit a decoder error.
    4. Emit a code point whose value is *code point*.
7. If *byte* is 0x7E, set hz-gb-2312 lead to 0x7E and continue.
8. If the hz-gb-2312 flag is set:
    1. If *byte* is in the range 0x20 to 0x7F, set hz-gb-2312 lead to *byte* and continue.
    2. If *byte* is 0x0A, unset the hz-gb-2312 flag.
    3. Emit a decoder error.
9. If *byte* is in the range 0x00 to 0x7F, emit a code point whose value is *byte*.
10. Emit a decoder error.

The **hz-gb-2312 encoder** (encoder for hz-gb-2312) is:

1. Let *code point* be code point pointer.
2. If *code point* is the EOF code point, emit the EOF byte.
3. Increase the code point pointer by one.
4. If *code point* is in the range U+0000 to U+007F and the hz-gb-2312 flag is set, decrease the code point pointer by one, unset the hz-gb-2312 flag, and emit two bytes 0x7E 0x7D.
5. If *code point* is 0x007E, emit two bytes 0x7E 0x7E.
6. If *code point* is in the range U+0000 to U+007F, emit a byte whose value is *code point*.
7. If the hz-gb-2312 flag is unset, decrease the code point pointer by one, set the hz-gb-2312 flag, and emit two bytes 0x7E 0x7B.
8. Let *pointer* be the index pointer for *code point* in index gbk.
9. If *pointer* is null, emit an encoder error.
10. Let *lead* be *pointer* / 190 + 1.
11. Let *trail* be *pointer* % 190 − 0x3F.
12. If either *lead* or *trail* is not in the range 0x21 to 0x7E, emit an encoder error.
13. Emit two bytes whose values are *lead* and *trail*.

10 Legacy multi-byte Chinese (traditional) encodings

10.1 big5

The **big5 lead** is initially 0x00.

The **big5 decoder** (decoder for big5) is:

1. Let *byte* be byte pointer.
2. If *byte* is the EOF byte and big5 lead is 0x00, emit the EOF code point.
3. If *byte* is the EOF byte and big5 lead is not 0x00, set big5 lead to 0x00 and emit a decoder error.
4. Increase the byte pointer by one.
5. If big5 lead is not 0x00, let *lead* be big5 lead, let *pointer* be null, set big5 lead to 0x00, and then run these substeps:
    1. Let *offset* be 0x40 if *byte* is less than 0x7F, or 0x62 otherwise.
    2. If *byte* is in the range 0x40 to 0x7E or 0xA1 to 0xFE, set *pointer* to (*lead* − 0x81) × 157 + (*byte* − *offset*).
    3. If there is a row in the table below whose first column is *pointer*, emit the *two* code points listed in its second column:

Pointer	Code points
1133	U+00CA U+0304
1135	U+00CA U+030C
1164	U+00EA U+0304
1166	U+00EA U+030C

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, or the index code point for *pointer* in index big5 otherwise.
    5. If *pointer* is null, decrease the byte pointer by one.
    6. If *code point* is null, emit a decoder error.
    7. Emit a code point whose value is *code point*.
6. If *byte* is in the range 0x00 to 0x7F, emit a code point whose value is *byte*.
7. If *byte* is in the range 0x81 to 0xFE, set big5 lead to *byte* and continue.
8. Emit a decoder error.

The **big5 encoder** (encoder for big5) is:

1. Let *code point* be code point pointer.
2. If *code point* is the EOF code point, emit the EOF byte.
3. Increase the code point pointer by one.
4. If *code point* is in the range U+0000 to U+007F, emit a byte whose value is *code point*.
5. Let *pointer* be the index pointer for *code point* in index big5.
6. If *pointer* is null, emit an encoder error.
7. Let *lead* be *pointer* / 157 + 0x81.
8. If *lead* is less than 0xA1, emit an encoder error.

Note: Avoid emitting Hong Kong Supplementary Character Set extensions literally.

9. Let *trail* be *pointer* % 157.
10. Let *offset* be 0x40 if *trail* is less than 0x3F, or 0x62 otherwise.

11. Emit two bytes whose values are *lead* and *trail* + *offset*.

11 Legacy multi-byte Japanese encodings

11.1 euc-jp

The **euc-jp first** and **euc-jp second** are 0x00.

The **euc-jp decoder** (decoder for euc-jp) is:

1. Let *byte* be byte pointer.
2. If *byte* is the EOF byte and euc-jp first and euc-jp second are 0x00, emit the EOF code point.
3. If *byte* is the EOF byte and either euc-jp first or euc-jp second is not 0x00, set euc-jp first and euc-jp second to 0x00, and emit a decoder error.
4. Increase the byte pointer by one.
5. If euc-jp second is not 0x00, let *lead* be euc-jp second, set euc-jp second to 0x00 and run these substeps:
    1. Let *code point* be null.
    2. If *lead* and *byte* are both in the range 0xA1 to 0xFE, set *code point* to the index code point for (*lead* − 0xA1) × 94 + *byte* − 0xA1 in index jis0212.
    3. If *byte* is not in the range 0xA1 to 0xFE, decrease byte pointer by one.
    4. If *code point* is null, emit a decoder error.
    5. Emit a code point whose value is *code point*.
6. If euc-jp first is 0x8E and *byte* is in the range 0xA1 to 0xDF, set euc-jp first to 0x00 and emit a code point whose value is 0xFF61 + *byte* − 0xA1.
7. If euc-jp first is 0x8F and *byte* is in the range 0xA1 to 0xFE, set euc-jp first to 0x00, euc-jp second to *byte*, and continue.
8. If euc-jp first is not 0x00, let *lead* be euc-jp first, set euc-jp first to 0x00, and run these substeps:
    1. Let *code point* be null.
    2. If *lead* and *byte* are both in the range 0xA1 to 0xFE, set *code point* to the index code point for (*lead* − 0xA1) × 94 + *byte* − 0xA1 in index jis0208.
    3. If *byte* is not in the range 0xA1 to 0xFE, decrease byte pointer by one.
    4. If *code point* is null, emit a decoder error.
    5. Emit a code point whose value is *code point*.
9. If *byte* is in the range 0x00 to 0x7F, emit a code point whose value is *byte*.
10. If *byte* is 0x8E, 0x8F, or in the range 0xA1 to 0xFE, set euc-jp first to *byte* and continue.
11. Emit a decoder error.

The **euc-jp encoder** (encoder for euc-jp) is:

1. Let *code point* be the code point pointer.
2. If *code point* is the EOF code point, emit the EOF byte.
3. Increase the code point pointer by one.
4. If *code point* is in the range U+0000 to U+007F, emit a byte whose value is *code point*.
5. If *code point* is U+00A5, emit byte 0x5C.
6. If *code point* is U+203E, emit byte 0x7E.
7. If *code point* is in the range U+FF61 to U+FF9F, emit two bytes whose values are 0x8E and *code point* − 0xFF61 + 0xA1.
8. Let *pointer* be the index pointer for *code point* in index jis0208.
9. If *pointer* is null, emit an encoder error.

10. Let *lead* be *pointer* / 94 + 0xA1.
11. Let *trail* be *pointer* % 94 + 0xA1.
12. Emit two bytes whose values are *lead* and *trail*.

Note: Contemporary implementations do not use index jis0212 for the euc-jp encoder.

11.2 iso-2022-jp

The **iso-2022-jp state** is initially **ASCII state**.

The **iso-2022-jp jis0212 flag** is initially unset.

The **iso-2022-jp lead** is initially 0x00.

The **iso-2022-jp decoder** (decoder for iso-2022-jp) is:

1. Let *byte* be byte pointer.
2. If *byte* is not the EOF byte, increase the byte pointer by one.
3. Based on iso-2022-jp state:

↪ **ASCII state**

Based on *byte*:

↪ **0x1B**

Set iso-2022-jp state to **escape start state** and continue.

↪ **0x00 to 0x7F**

Emit a code point whose value is *byte*.

↪ **EOF byte**

Emit the EOF code point.

↪ **Otherwise**

Emit a decoder error.

↪ **Escape start state**

1. If *byte* is either 0x24 or 0x28, set iso-2022-jp lead to *byte*, iso-2022-jp state to **escape middle state**, and continue.
2. If *byte* is not the EOF byte, decrease the byte pointer by one.
3. Set iso-2022-jp state to **ASCII state** and emit a decoder error.

↪ **Escape middle state**

1. Let *lead* be iso-2022-jp lead and set iso-2022-jp lead to 0x00.
2. If *lead* is 0x24 and *byte* is either 0x40 or 0x42, unset the iso-2022-jp jis0212 flag, set iso-2022-jp state to **lead state**, and continue.
3. If *lead* is 0x24 and *byte* is 0x28, set iso-2022-jp state to **escape final state** and continue.
4. If *lead* is 0x28 and *byte* is either 0x42 or 0x4A, set iso-2022-jp state to **ASCII state** and continue.
5. If *lead* is 0x28 and *byte* is 0x49, set iso-2022-jp state to **Katakana state** and continue.
6. If *byte* is the EOF byte, decrease byte pointer by one, or decrease it by two otherwise.
7. Set iso-2022-jp state to **ASCII state** and emit a decoder error.

↪ **Escape final state**

1. If *byte* is 0x44, set the iso-2022-jp jis0212 flag, set iso-2022-jp state to **lead state**, and continue.
2. If *byte* is the EOF byte, decrease byte pointer by two, or decrease it by three otherwise.
3. Set iso-2022-jp state to **ASCII state** and emit a decoder error.

↪ **Lead state**

Based on *byte*:

↪ **0x0A**
Set iso-2022-jp state to **ASCII state** and emit code point U+000A.

↪ **0x1B**
Set iso-2022-jp state to **escape start state** and continue.

↪ **EOF byte**
Emit the EOF code point.

↪ **Otherwise**
Set iso-2022-jp lead to *byte*, iso-2022-jp state to **trail state**, and continue.

↪ **Trail state**

1. Set the iso-2022-jp state to **lead state**.
2. If *byte* is the EOF byte, emit a decoder error.
3. Let *code point* be null and let *pointer* be (iso-2022-jp lead − 0x21) × 94 + *byte* − 0x21.
4. If iso-2022-jp lead and *byte* are both in the range 0x21 to 0x7E, set *code point* to the index code point for *pointer* in index jis0208 if the iso-2022-jp jis0212 flag is unset, or in index jis0212 otherwise.
5. If *code point* is null, emit a decoder error.
6. Emit a code point whose value is *code point*.

↪ **Katakana state**

Based on *byte*:

↪ **0x1B**
Set iso-2022-jp state to **escape start state** and continue.

↪ **0x21 to 0x5F**
Emit a code point whose value is 0xFF61 + *byte* − 0x21.

↪ **EOF byte**
Emit the EOF code point.

↪ **Otherwise**
Emit a decoder error.

The **iso-2022-jp encoder** (encoder for iso-2022-jp) is:

1. Let *code point* be code point pointer.
2. If *code point* is the EOF code point, emit the EOF byte.
3. Increase the code point pointer by one.
4. If *code point* is in the range U+0000 to U+007F, or is U+00A5 or U+203E, and iso-2022-jp state is not **ASCII state**, decrease the code point pointer by one, set iso-2022-jp state to **ASCII state**, and emit three bytes 0x1B 0x28 0x42.
5. If *code point* is in the range U+0000 to U+007F, emit a byte whose value is *code point*.
6. If *code point* is U+00A5, emit byte 0x5C.
7. If *code point* is U+203E, emit byte 0x7E.
8. If *code point* is in the range U+FF61 to U+FF9F and iso-2022-jp state is not **Katakana state**, decrease the code point pointer by one, set iso-2022-jp state to **Katakana state**, and emit three bytes 0x1B 0x28 0x49.
9. If *code point* is in the range U+FF61 to U+FF9F, emit a byte whose value is *code point* − 0xFF61 + 0x21.
10. If iso-2022-jp state is not **lead state**, decrease the code point pointer by one, set iso-2022-jp state to **lead state**, and emit three bytes 0x1B 0x24 0x42.
11. Let *pointer* be the index pointer for *code point* in index jis0208.
12. If *pointer* is null, emit an encoder error.

13. Let *lead* be *pointer* / 94 + 0x21.
14. Let *trail* be *pointer* % 94 + 0x21.
15. Emit two bytes whose values are *lead* and *trail*.

11.3 shift_jis

The **shift_jis lead** is initially 0x00.

The **shift_jis decoder** (decoder for shift_jis) is:

1. Let *byte* be byte pointer.
2. If *byte* is the EOF byte and shift_jis lead is 0x00, emit the EOF code point.
3. If *byte* is the EOF byte, shift_jis lead is not 0x00, set shift_jis lead to 0x00 and emit a decoder error.
4. Increase the byte pointer by one.
5. 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 run these substeps:
    1. Let *offset* be 0x40 if *byte* is less than 0x7F, or 0x41 otherwise.
    2. Let *lead offset* be 0x81 if *lead* is less than 0xA0, or 0xC1 otherwise.
    3. If *byte* is in the range 0x40 to 0x7E or 0x80 to 0xFC, set *pointer* to (*lead* − *lead offset*) × 188 + *byte* − *offset*.
    4. Let *code point* be null if *pointer* is null, or the index code point for *pointer* in index jis0208 otherwise.
    5. If *pointer* is null, decrease the byte pointer by one.
    6. If *code point* is null, emit a decoder error.
    7. Emit a code point whose value is *code point*.
6. If *byte* is in the range 0x00 to 0x80, emit a code point whose value is *byte*.
7. If *byte* is in the range 0xA1 to 0xDF, emit a code point whose value is 0xFF61 + *byte* − 0xA1.
8. If *byte* is in the range 0x81 to 0x9F or 0xE0 to 0xFC, set shift_jis lead to *byte* and continue.
9. Emit a decoder error.

The **shift_jis encoder** (encoder for shift_jis) is:

1. Let *code point* be the code point pointer.
2. If *code point* is the EOF code point, emit the EOF byte.
3. Increase the code point pointer by one.
4. If *code point* is in the range U+0000 to U+0080, emit a byte whose value is *code point*.
5. If *code point* is U+00A5, emit byte 0x5C.
6. If *code point* is U+203E, emit byte 0x7E.
7. If *code point* is in the range U+FF61 to U+FF9F, emit a byte whose value is *code point* − 0xFF61 + 0xA1.
8. Let *pointer* be the index pointer for *code point* in index jis0208.
9. If *pointer* is null, emit an encoder error.
10. Let *lead* be *pointer* / 188.
11. Let *lead offset* be 0x81 if *lead* is less than 0x1F, or 0xC1 otherwise.
12. Let *trail* be *pointer* % 188.
13. Let *offset* be 0x40 if *trail* is less than 0x3F, or 0x41 otherwise.
14. Emit two bytes whose values are *lead* + *lead offset* and *trail* + *offset*.

12 Legacy multi-byte Korean encodings

12.1 euc-kr

The **euc-kr lead** is initially 0x00.

The **euc-kr decoder** (decoder for euc-kr) is:

1. Let *byte* be byte pointer.
2. If *byte* is the EOF byte and euc-kr lead is 0x00, emit the EOF code point.
3. If *byte* is the EOF byte and euc-kr lead is not 0x00, set euc-kr lead to 0x00 and emit a decoder error.
4. Increase the byte pointer by one.
5. 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 run these substeps:
    1. If *lead* is in the range 0x81 to 0xC6, let *temp* be (26 + 26 + 126) × (*lead* − 0x81), and then set *pointer* to the result of the equation below, depending on *byte*:

        ↪ **0x41 to 0x5A**
        : *temp* + *byte* − 0x41

        ↪ **0x61 to 0x7A**
        : *temp* + 26 + *byte* − 0x61

        ↪ **0x81 to 0xFE**
        : *temp* + 26 + 26 + *byte* − 0x81

    2. If *lead* is in the range 0xC7 to 0xFE and *byte* is in the range 0xA1 to 0xFE, set *pointer* to (26 + 26 + 126) × (0xC7 − 0x81) + (*lead* − 0xC7) × 94 + (*byte* − 0xA1).
    3. Let *code point* be null if *pointer* is null, or the index code point for *pointer* in index euc-kr otherwise.
    4. If *pointer* is null, decrease the byte pointer by one.
    5. If *code point* is null, emit a decoder error.
    6. Emit a code point whose value is *code point*.
6. If *byte* is in the range 0x00 to 0x7F, emit a code point whose value is *byte*.
7. If *byte* is in the range 0x81 to 0xFE, set euc-kr lead to *byte* and continue.
8. Emit a decoder error.

The **euc-kr encoder** (encoder for euc-kr) is:

1. Let *code point* be the code point pointer.
2. If *code point* is the EOF code point, emit the EOF byte.
3. Increase the code point pointer by one.
4. If *code point* is in the range U+0000 to U+007F, emit a byte whose value is *code point*.
5. Let *pointer* be the index pointer for *code point* in index euc-kr.
6. If *pointer* is null, emit an encoder error.
7. If *pointer* is less than (26 + 26 + 126) × (0xC7 − 0x81), run these substeps:
    1. Let *lead* be *pointer* / (26 + 26 + 126) + 0x81.
    2. Let *trail* be *pointer* % (26 + 26 + 126).
    3. Let *offset* be 0x41 if *trail* is less than 26, 0x47 if *trail* is less than 26 + 26, or 0x4D otherwise.
    4. Emit two bytes whose values are *lead* and *trail* + *offset*.
8. Set *pointer* to *pointer* − (26 + 26 + 126) × (0xC7 − 0x81).
9. Let *lead* be *pointer* / 94 + 0xC7.

10. Let *trail* be *pointer* % 94 + 0xA1.
11. Emit two bytes whose values are *lead* and *trail*.

12.2 iso-2022-kr

The **iso-2022-kr state** is initially **ASCII state**.

The **iso-2022-kr lead** is initially 0x00.

The **iso-2022-kr decoder** (decoder for iso-2022-kr) is:

1. Let *byte* be byte pointer.
2. If *byte* is not the EOF byte, increase the byte pointer by one.
3. Based on iso-2022-kr state:

↪ **ASCII state**
Based on *byte*:

↪ **0x0E**
Set iso-2022-kr state to **lead state** and continue.

↪ **0x0F**
Continue.

↪ **0x1B**
Set iso-2022-kr state to **escape start state** and continue.

↪ **0x00 to 0x7F**
Emit a code point whose value is *byte*.

↪ **EOF byte**
Emit the EOF code point.

↪ **Otherwise**
Emit a decoder error.

↪ **Escape start state**

1. If *byte* is 0x24, set iso-2022-kr state to **escape middle state** and continue.
2. If *byte* is not the EOF byte, decrease the byte pointer by one.
3. Set iso-2022-kr state to **ASCII state** and emit a decoder error.

↪ **Escape middle state**

1. If *byte* is 0x29, set iso-2022-kr state to **escape end state** and continue.
2. If *byte* is the EOF byte, decrease the byte pointer by one, or decrease it by two otherwise.
3. Set iso-2022-kr state to **ASCII state** and emit a decoder error.

↪ **Escape end state**

1. If *byte* is 0x43, set iso-2022-kr state to **ASCII state** and continue.
2. If *byte* is the EOF byte, decrease the byte pointer by two, or decrease it by three otherwise.
3. Set iso-2022-kr state to **ASCII state** and emit a decoder error.

↪ **Lead state**
Based on *byte*:

↪ **0x0A**
Set iso-2022-kr state to **ASCII state** and emit code point U+000A.

↪ **0x0E**
Continue.

↪ **0x0F**
: Set iso-2022-kr state to **ASCII state** and continue.

↪ **EOF byte**
: Emit the EOF code point.

↪ **Otherwise**
: Set iso-2022-kr lead to *byte*, set iso-2022-kr state to **trail state**, and continue.

↪ **Trail state**

1. Set the iso-2022-kr state to **lead state**.
2. If *byte* is the EOF byte, emit a decoder error.
3. Let *code point* be null.
4. If iso-2022-kr lead is in the range 0x21 to 0x46 and *byte* is in the range 0x21 to 0x7E, set *code point* to the index code point for (26 + 26 + 126) × (iso-2022-kr lead − 1) + 26 + 26 + *byte* − 1 in index euc-kr.
5. If iso-2022-kr lead is in the range 0x47 to 0x7E and *byte* is in the range 0x21 to 0x7E, set *code point* to the index code point for (26 + 26 + 126) × (0xC7 − 0x81) + (iso-2022-kr lead − 0x47) × 94 + (*byte* − 0x21) in index euc-kr.
6. If *code point* is not null, emit *code point*.
7. Emit a decoder error.

The **iso-2022-kr initialization flag** is initially unset.

The **iso-2022-kr encoder** (encoder for iso-2022-kr) is:

1. Let *code point* be code point pointer.
2. If *code point* is the EOF code point and iso-2022-kr state is not **ASCII state**, set iso-2022-kr state to **ASCII state** and emit byte 0x0F.
3. If *code point* is the EOF code point, emit the EOF byte.
4. If the iso-2022-kr initialization flag is unset, set the iso-2022-kr initialization flag and emit four bytes 0x1B 0x24 0x29 0x43.
5. Increase the code point pointer by one.
6. If *code point* is in the range U+0000 to U+007F and iso-2022-kr state is not **ASCII state**, decrease the code point pointer by one, set iso-2022-kr state to **ASCII state**, and emit byte 0x0F.
7. If *code point* is in the range U+0000 to U+007F, emit a byte whose value is *code point*.
8. If iso-2022-kr state is not **lead state**, decrease the code point pointer by one, set iso-2022-kr state to **lead state**, and emit byte 0x0E.
9. Let *pointer* be the index pointer for *code point* in index euc-kr.
10. If *pointer* is null, emit an encoder error.
11. If *pointer* is less than (26 + 26 + 126) × (0xC7 − 0x81), run these substeps:
    1. Let *lead* be *pointer* / (26 + 26 + 126) + 1.
    2. Let *trail* be *pointer* % (26 + 26 + 126) − 26 − 26 + 1.
    3. If *lead* is not in the range 0x21 to 0x46 or *trail* is not in the range 0x21 to 0x7E, emit an encoder error.
    4. Emit two bytes whose values are *lead* and *trail*.
12. Set *pointer* to *pointer* − (26 + 26 + 126) × (0xC7 − 0x81).
13. Let *lead* be *pointer* / 94 + 0x47.
14. Let *trail* be *pointer* % 94 + 0x21.
15. If *lead* is not in the range 0x47 to 0x7E or *trail* is not in the range 0x21 to 0x7E, emit an encoder error.

16. Emit two bytes whose values are *lead* and *trail*.

13 Legacy utf-16 encodings

> ***Note: Contrary to the Unicode standard, checking for a byte order mark happens before an encoding to decode a byte stream is chosen.***

13.1 utf-16

The **utf-16 lead byte** and **utf-16 lead surrogate** are initially null and the **utf-16be flag** is initially unset.

The **utf-16 decoder** (decoder for utf-16) is:

1. Let *byte* be byte pointer.
2. If *byte* is the EOF byte and utf-16 lead byte and utf-16 lead surrogate are null, emit the EOF code point.
3. If *byte* is the EOF byte and either utf-16 lead byte or utf-16 lead surrogate is not null, set utf-16 lead byte and utf-16 lead surrogate to null, and emit a decoder error.
4. Increase the byte pointer by one.
5. If utf-16 lead byte is null, set utf-16 lead byte to *byte* and continue.
6. Let *code point* be the result of:

   ↪ **utf-16be flag is set**

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

   ↪ **utf-16be flag is unset**

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

   Then set utf-16 lead byte to null.
7. If utf-16 lead surrogate is not null, let *lead surrogate* be utf-16 lead surrogate, set utf-16 lead surrogate to null, and then run these substeps:
   1. If *code point* is in the range U+DC00 to U+DFFF, emit a code point whose value is 0x10000 + (*lead surrogate* − 0xD800) × 0x400 + (*code point* − 0xDC00).
   2. Decrease the byte pointer by two and emit a decoder error.
8. If *code point* is in the range U+D800 to U+DBFF, set utf-16 lead surrogate to *code point* and continue.
9. If *code point* is in the range U+DC00 to U+DFFF, emit a decoder error.
10. Emit code point *code point*.

To **convert a *code unit* to bytes** run these steps:

1. Let *byte1* be *code unit* >> 8.
2. Let *byte2* be *code unit* & 0x00FF.
3. Then return the bytes in order:

   ↪ **utf-16be flag is set**

   byte1, then *byte2*.

   ↪ **utf-16be flag is unset**

   byte2, then *byte1*.

The **utf-16 encoder** (encoder for utf-16) is:

1. Let *code point* be code point pointer.
2. If *code point* is in the range 0xD800 to 0xDFFF, emit an encoder error.
3. If *code point* is the EOF code point, emit the EOF byte.
4. Increase the code point pointer by one.

5. If *code point* is in the range 0x00 to 0xFFFF, emit the sequence resulting of converting *code point* to bytes.
6. Let *lead* be (*code point* − 0x10000) / 0x400 + 0xD800, converted to bytes.
7. Let *trail* be (*code point* − 0x10000) % 0x400 + 0xDC00, converted to bytes.
8. Emit a sequence of bytes that consists of *lead* followed by *trail*.

13.2 utf-16be

The **utf-16be decoder** (decoder for utf-16be) is the utf-16 decoder with the utf-16be flag set.

The **utf-16be encoder** (encoder for utf-16be) is the utf-16 encoder with the utf-16be flag set.

References

[RFC2119]

Key words for use in RFCs to Indicate Requirement Levels, Scott Bradner. IETF.

[UNICODE]

Unicode Standard. Unicode Consortium.

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 specification. Likewise people have helped making this specification what it is today.

Ideally they are all listed here so please contact the editor with any omissions.

With that, many thanks to Charles McCathieNeville, David Carlisle, Doug Ewell, Erik van der Poel, 譚永鋒 (Frank Yung-Fong Tang), Ian Hickson, Joshua Bell, 신정식 (Jungshik Shin), Ken Lunde, Leif Halvard Silli, Makoto Kato, Mark Callow, Mark Davis, Martin Dürst, Masatoshi Kimura, Ms2ger, Norbert Lindenberg, Øistein E. Andersen, Peter Krefting, Philip Jägenstedt, Philip Taylor, Shawn Steele, Simon Montagu, Simon Pieters, and 成瀬ゆい (Yui Naruse) for being awesome.