Chapter 3

Conformance

This chapter defines conformance to the Unicode Standard in terms of the principles and encoding architecture it embodies. The first section consists of the conformance clauses, followed by sections that define more precisely the technical terms used in those clauses. The remaining sections contain the formal algorithms that are part of conformance and referred to by the conformance clause. These algorithms specify the required results rather than the specific implementation; all implementations that produce results identical to the results of the formal algorithms are conformant.

In this chapter, conformance subclauses are identified with a letter *C*. Definitions are identified with the letter *D*. Bulleted items are explanatory comments regarding definitions or subclauses.

Except for *Section 3.12, Bidirectional Behavior*, the numbering of rules and definitions matches that of *The Unicode Standard, Version 2.0*. Where new rules and definitions were added, letters are used with numbers—for example, *D7a*.

3.1 Conformance Requirements

This section specifies the formal conformance requirements for processes implementing Version 3.0 of the Unicode Standard. Note that this clause has been revised from the previous versions of the Unicode Standard. These revisions do not change the substance of the conformance requirements previously set forth, but rather are formalized and extended to allow for the use of transformation formats. Implementations that satisfied the conformance clause of the previous versions of the Unicode Standard will satisfy this revised clause.

Byte Ordering

C1 A process shall interpret Unicode code values as 16-bit quantities.

- Unicode values can be stored in native 16-bit machine words.
- For information on use of `wchar_t` or other programming language types to represent Unicode values, see *Section 5.2, ANSI/ISO C wchar_t*.

C2 The Unicode Standard does not specify any order of bytes inside a Unicode value.

- Machine architectures differ in *ordering* in terms of whether the most significant byte or the least significant byte comes first. These sequences are known as "big-endian" and "little-endian" orders, respectively.

C3 A process shall interpret a Unicode value that has been serialized into a sequence of bytes by most significant byte first, in the absence of higher-level protocols.

- The majority of all interchange occurs with processes running on the same or a similar configuration. As a result, intradomain interchange of Unicode text in the

domain-specific byte order is fully conformant and limits the role of the canonical byte order to interchange of Unicode text across domain, or where the nature of the originating domain is unknown. (For a discussion of the use of *byte order mark* to indicate byte orderings, see *Section 2.7, Special Character and Noncharacter Values.*)

Invalid Code Values

C4 A process shall not interpret an unpaired high- or low-surrogate as an abstract character.

C5 A process shall not interpret either U+FFFE or U+FFFF as an abstract character.

C6 A process shall not interpret any unassigned code value as an abstract character.

- These clauses do not preclude the assignment of certain generic semantics (for example, rendering with a glyph to indicate the character block) that allow for graceful behavior in the presence of code values that are outside a supported subset or code values that are unpaired surrogates.
- Private-use code values are assigned, but can be given any interpretation by conformant processes.

Interpretation

C7 A process shall interpret a coded character representation according to the character semantics established by this standard, if that process does interpret that coded character representation.

- This restriction does not preclude internal transformations that are never visible external to the process.

C8 A process shall not assume that it is required to interpret any particular coded character representation.

- Any means for specifying a subset of characters that a process can interpret is outside the scope of this standard.
- The semantics of a code value in the Private Use Area is outside the scope of this standard.
- Although these clauses are not intended to preclude enumerations or specifications of the characters that a process or system is able to interpret, they do separate supported subset enumerations from the question of conformance. In real life, any system may occasionally receive an unfamiliar character code that it is unable to interpret.

C9 A process shall not assume that the interpretations of two canonical-equivalent character sequences are distinct.

- Ideally, an implementation would always interpret two canonical-equivalent character sequences identically. There are practical circumstances under which implementations may reasonably distinguish them.
- Even processes that normally do not distinguish between canonical-equivalent character sequences can have reasonable exception behavior. Some examples of this behavior include graceful fallback processing by processes unable to support correct positioning of nonspacing marks; “Show Hidden Text” modes that reveal memory representation structure; and the choice of ignoring collating behavior of

combining sequences that are not part of the repertoire of a specified language (see *Section 5.13, Strategies for Handling Nonspacing Marks*).

Modification

C10 A process shall make no change in a valid coded character representation other than the possible replacement of character sequences by their canonical-equivalent sequences, if that process purports not to modify the interpretation of that coded character representation.

- Replacement of a character sequence by a compatibility-equivalent sequence does modify the interpretation of the text.
- Replacement or deletion of a character sequence that the process cannot or does not interpret does modify the interpretation of the text.
- Changing the bit or byte ordering when transforming between different machine architectures does not modify the interpretation of the text.
- Transforming to a different encoding form does not modify the interpretation of the text.

Transformations

C11 When a process interprets a byte sequence in a Unicode Transformation Format, it shall interpret that byte sequence in accordance with the character semantics established by this standard for the corresponding Unicode character sequence.

C12 When a process generates data in a Unicode Transformation Format, it shall not emit ill-formed byte sequences. When a process interprets data in a Unicode Transformation Format, it shall treat illegal byte sequences as an error condition.

Bidirectional Text

C13 A process that displays text containing supported right-to-left characters or embedding codes shall display all visible representations of characters (excluding format characters) in the same order as if the bidirectional algorithm had been applied to the text, in the absence of higher-level protocols (see Section 3.12, Bidirectional Behavior).

Unicode Technical Reports

The following technical reports are approved and considered part of Version 3.0 of the Unicode Standard. These reports may contain either normative or informative material, or both. Any reference to Version 3.0 of the standard automatically includes these technical reports.

- UTR #11: East Asian Width, Version 5.0
- UTR #13: Unicode Newline Guidelines, Version 5.0
- UTR #14: Line Breaking Properties, Version 6.0
- UTR #15: Unicode Normalization Forms, Version 18.0

3.2 Semantics

This and the following sections more precisely define the terms that are used in the conformance clauses.

D1 *Normative properties and behavior:* The following are normative character properties and normative behavior of the Unicode Standard:

1. Simple properties
2. Character combination
3. Canonical decomposition
4. Compatibility decomposition
5. Surrogate property
6. Canonical ordering behavior
7. Bidirectional behavior, as interpreted according to the Unicode bidirectional algorithm
8. Conjoining jamo behavior, as interpreted according to *Section 3.11, Conjoining Jamo Behavior*

D2 *Character semantics:* The semantics of a character are established by its character name, representative glyph, and normative properties and behavior.

- A character may have a broader range of use than the most literal interpretation of its name might indicate; the coded representation, name, and representative glyph need to be taken in context when establishing the semantics of a character. For example, U+002E FULL STOP can represent a sentence period, an abbreviation period, a decimal number separator in English, a thousands number separator in German, and so on.
- Consistency with the representative glyph does not require that the images be identical or even graphically similar; rather, it means that both images are generally recognized to be representations of the same character. Representing the character U+0061 LATIN SMALL LETTER A by the glyph "X" would violate its character identity.
- Some normative behavior is default behavior; this behavior can be overridden by higher-level protocols. However, in the absence of such protocols, the behavior must be observed so as to follow the character semantics.
- The character combination properties and the canonical ordering behavior cannot be overridden by higher-level protocols.

3.3 Characters and Coded Representations

D3 *Abstract character:* a unit of information used for the organization, control, or representation of textual data.

- When representing data, the nature of that data is generally symbolic as opposed to some other kind of data (for example, numeric, aural, or visual). Examples of such symbolic data include letters, ideographs, digits, punctuation, technical symbols, and dingbats.

- An abstract character has no concrete form and should not be confused with a *glyph*.
- An abstract character does not necessarily correspond to what a user thinks of as a "character" and should not be confused with a *grapheme*.
- The abstract characters encoded by the Unicode Standard are known as Unicode abstract characters.
- Abstract characters not directly encoded by the Unicode Standard can be represented by the use of combining character sequences.

D4 *Abstract character sequence:* an ordered sequence of abstract characters.

Each Unicode abstract character is encoded one or more times. Each encoding, which consists of the relationship between an abstract character and its scalar value (see D28), is called an *encoded character*. Each scalar value is represented in either of two ways: as a single code value or as a sequence of two surrogate code values.

D5 *Code value:* the minimal bit combination that can represent a unit of encoded text for processing or interchange.

- Other character encoding standards typically use code values defined as 8-bit units. The same is true for the UTF-8 transformation format of the Unicode Standard. However, the code values used in the UTF-16 form of the Unicode Standard are 16-bit units. These 16-bit code values are also known simply as *Unicode values*.
- A c*ode value* is also referred to as a *code unit* in the information industry.
- A single abstract character may correspond to more than one code value—for example, U+00C5 Å LATIN CAPITAL LETTER A WITH RING and U+212B Å ANGSTROM SIGN.
- Multiple code values may be required to represent a single abstract character. For example, a byte is the code unit in SJIS: characters such as "a" can be represented with a single code value, whereas ideographs require two code values.
- In some encodings, specific code units cannot be used to represent an encoded character in isolation. This restriction includes a single surrogate (high or low) in Unicode, the bytes 80–FF in UTF-8, or the bytes 81–9F, E0–EF in SJIS.

D6 *Coded character representation:* an ordered sequence of one or more code values that is associated with an abstract character in a given character repertoire.

- A Unicode abstract character is generally encoded by a single Unicode code value; the only exception involves surrogate pairs (which are provided for future extension, but are not currently used to represent any abstract characters).

D7 *Coded character sequence:* an ordered sequence of coded character representations.

Unless specified otherwise for clarity, in the text of the Unicode Standard the term *character* alone generally designates a coded character representation. Similarly, the term *character sequence* alone generally designates a coded character sequence.

D7a *Deprecated character:* a coded character whose use is strongly discouraged. Such characters are retained in the standard, but should not be used.

- Deprecated characters are retained in the standard so that previously conforming data stay conformant in future versions of the standard. Deprecated characters are to be distinguished from obsolete characters.
- Obsolete characters are historical. They do not occur in modern text, but they are not deprecated; their use is not discouraged.

D8 *Higher-level protocol:* any agreement on the interpretation of Unicode characters that extends beyond the scope of this standard. Such an agreement need not be formally announced in data; it may be implicit in the context.

3.4 Simple Properties

The Unicode Standard, Version 3.0, defines the normative simple character properties of *case*, *numeric value*, *directionality,* and *mirrored*. *Chapter 4, Character Properties*, contains explicit mappings of characters to character properties. These mappings represent the default properties for conformant processes in the absence of explicit, overriding, higher-level protocols. Additional properties that are specific to particular characters (such as the definition and use of the *right-to-left override* character or *zero-width spaces*) are discussed in the relevant sections of this standard.

- The *Unicode Character Database* contains additional properties, such as category and case mappings, that are informative rather than normative.

The interpretation of some properties (such as the case of a character) is independent of context, whereas the interpretation of others (such as directionality) is applicable to a character sequence as a whole, rather than to the individual characters that compose the sequence.

D9 *Directionality property:* a property of every graphic character that determines its horizontal ordering as specified in *Section 3.12, Bidirectional Behavior*.

- Interpretation of directional properties according to the Unicode bidirectional algorithm is needed for layout of right-to-left scripts such as Arabic and Hebrew.

D10 *Mirrored property:* the property of characters whose images are mirrored horizontally in text that is laid out from right to left (versus left to right). (See *Section 4.7, Mirrored—Normative*.)

- In other words, U+0028 LEFT PARENTHESIS is interpreted as an opening parenthesis; in a left-to-right context, this character will appear as "("; in a right-to-left context, it will be mirrored and appear as ")".

- This is the default behavior in Unicode text. (For more information, see the "Semantics of Paired Punctuation" subsection in *Section 6.1, General Punctuation*.)

D10a *Case property:* a property of characters in certain alphabets whereby certain characters are considered variants of a single letter. (See *Section 4.1, Case—Normative*.)

D10b *Numeric value property:* a property of characters used to represent numbers. (See *Section 4.6, Numeric Value—Normative*.)

D11 *Special character properties:* The behavior of most characters does not require special attention in this standard. However, certain characters exhibit special behavior, which is described in the character block descriptions. These characters are listed in *Section 3.9, Special Character Properties*.

D12 *Private use:* Unicode values from U+E000 to U+F8FF and surrogate pairs (see *Section 3.7, Surrogates*) whose high-surrogate is from U+DB80 to U+DBFF are available for private use.

3.5 Combination

D13 *Base character:* a character that does not graphically combine with preceding characters, and that is neither a control nor a format character.

- Most Unicode characters are base characters. This sense of graphic combination does not preclude the presentation of base characters from adopting different contextual forms or participating in ligatures.

D14 *Combining character:* a character that graphically combines with a preceding base character. The combining character is said to *apply* to that base character.

- These characters are not used in isolation (unless they are being described). They include such characters as accents, diacritics, Hebrew points, Arabic vowel signs, and Indic matras.
- Even though a combining character is intended to be presented in graphical combination with a base character, circumstances may arise where either (1) no base character precedes the combining character or (2) a process is unable to perform graphical combination. In both cases, a process may present a combining character without graphical combination; that is, it may present it as if it were a base character.
- The representative images of combining characters are depicted with a dotted circle in the code charts; when presented in graphical combination with a preceding base character, that base character is intended to appear in the position occupied by the dotted circle.
- Combining characters generally take on the properties of their base character, while retaining their combining property.
- Control and format characters, such as *tab* or *right-left mark*, are not base characters. Combining characters do not apply to them.

D15 *Nonspacing mark:* a combining character whose positioning in presentation is dependent on its base character. It generally does not consume space along the visual baseline in and of itself.

- Such characters may be large enough to affect the placement of their base character relative to preceding and succeeding base characters. For example, a circumflex applied to an "i" may affect spacing ("î"), as might the character U+20DD COMBINING ENCLOSING CIRCLE.

D16 *Spacing mark:* a combining character that is not a nonspacing mark.

- Examples include U+093F DEVANAGARI VOWEL SIGN I. In general, the behavior of spacing marks does not differ greatly from that of base characters.

D17 *Combining character sequence:* a character sequence consisting of either a base character followed by a sequence of one or more combining characters, or a sequence of one or more combining characters.

- A combining character sequence is also referred to as a *composite character sequence.*

D17a *Defective combining character sequence:* a combining character sequence that does not start with a base character.

- Defective combining character sequences occur when a sequence of combining characters appears at the start of a string or follows a control or format character.

3.6 Decomposition

D18 *Decomposable character:* a character that is equivalent to a sequence of one or more other characters, according to the decomposition mappings found in the names list of *Section 14.1, Character Names List*. It may also be known as a *precomposed* character or *composite* character.

D19 *Decomposition:* a sequence of one or more characters that is equivalent to a decomposable character. A full decomposition of a character sequence results from decomposing each of the characters in the sequence until no characters can be further decomposed.

Compatibility Decomposition

D20 *Compatibility decomposition:* the decomposition of a character that results from recursively applying *both* the compatibility mappings *and* the canonical mappings found in the names list of *Section 14.1, Character Names List*, and those described in *Section 3.11, Conjoining Jamo Behavior*, until no characters can be further decomposed, and then reordering nonspacing marks according to *Section 3.10, Canonical Ordering Behavior*.

- A compatibility decomposition may remove formatting information.

D21 *Compatibility character:* a character that has a compatibility decomposition.

- Compatibility characters are included in the Unicode Standard to represent distinctions in other base standards. They support transmission and processing of legacy data. Their use is discouraged other than for legacy data.
- Replacing a compatibility character by its decomposition may lose round-trip convertibility with a base standard.

D22 *Compatibility equivalent:* Two character sequences are said to be compatibility equivalents if their full compatibility decompositions are identical.

Canonical Decomposition

D23 *Canonical decomposition:* the decomposition of a character that results from recursively applying the canonical mappings found in the names list of *Section 14.1, Character Names List*, and those described in *Section 3.11, Conjoining Jamo Behavior*, until no characters can be further decomposed, and then reordering nonspacing marks according to *Section 3.10, Canonical Ordering Behavior*.

- The canonical mappings are a subset of the compatibility mappings; however, a canonical decomposition does not remove formatting information.

D24 *Canonical equivalent:* Two character sequences are said to be canonical equivalents if their full canonical decompositions are identical.

- For example, the sequences <*o*, *combining-diaeresis*> and <*ö*> are canonical equivalents. Canonical equivalence is a Unicode property. It should not be confused with language-specific collation or matching, which may add additional equivalencies. For example, in Swedish, *ö* is treated as a completely different letter from *o*, collated after *z*. In German, *ö* is weakly equivalent to *oe* and collated with *oe*. In English, *ö* is just an *o* with a diacritic that indicates that it is pronounced separately from the previous letter (as in *coöperate*) and is collated with *o*.

Note: For definitions of *canonical composition* and *compatibility composition*, see Unicode Technical Report #15, "Unicode Normalization Forms," on the CD-ROM or the up-to-date version on the Unicode Web site.

3.7 Surrogates

D25 *High-surrogate:* a Unicode code value in the range U+D800 through U+DBFF.

D26 *Low-surrogate:* a Unicode code value in the range U+DC00 through U+DFFF.

D27 *Surrogate pair:* a coded character representation for a single abstract character that consists of a sequence of two Unicode values, where the first value of the pair is a high-surrogate and the second is a low-surrogate.

- Unlike combining characters, which have independent semantics and properties, high- and low-surrogates have no interpretation when they do not appear as part of a surrogate pair.
- Surrogate pairs are designed to allow representation of characters in future extensions of the Unicode Standard. There are no such currently assigned characters in this version of the standard, but it is widely expected that such characters will be added in the not too distant future. For more information, see *Section 13.4, Surrogates Area*, and *Section 5.4, Handling Surrogate Pairs*.

D28 *Unicode scalar value:* a number N from 0 to $10FFFF_{16}$ defined by applying the following algorithm to a character sequence S:

$N = U$	If S is a single, nonsurrogate value <U>
$N = (H - D800_{16}) * 400_{16} + (L - DC00_{16}) + 10000_{16}$	If S is a surrogate pair <H, L>

- Unicode scalar values are defined for use by standards such as SGML, XML, and HTML, which require a scalar value associated with abstract characters.
- This algorithm is identical with the ISO/IEC 10646 algorithm used to transform UTF-16 into UCS-4 (for more information, see *Appendix C, Relationship to ISO/IEC 10646*).
- The reverse mapping from the Unicode scalar value to a surrogate pair is given by

  $H = (S - 10000_{16}) / 400_{16} + D800_{16}$

  $L = (S - 10000_{16}) \% 400_{16} + DC00_{16}$

  The operators "/" and "%" are as defined in *Section 0.2, Notational Conventions.*
- A Unicode scalar value is also referred to as a *code position* or a *code point* in the information industry.

3.8 Transformations

More than one representation of Unicode data can be conformant to the Unicode Standard. Chief among them is UTF-8, discussed in *Section 2.3, Encoding Forms*, and *Appendix C.3, UCS Transformation Formats*. In addition, there are compression transformations such as the one described in the Unicode Technical Report #6, "A Standard

Compression Scheme for Unicode" on the CD-ROM or the up-to-date version on the Unicode Web site.

D29 A *Unicode* (or *UCS*) *transformation format* (UTF) transforms each Unicode scalar value into a unique sequence of code values. A UTF may specify a byte order for the serialization of the code values into bytes. A UTF may also specify the use of a *byte order mark*.

- Code values are particular units of computer storage specified by the transformation format—for example, 16-bit integers or bytes. In the latter case, a code value sequence can be referred to as a *byte sequence*.
- Any sequence of code values that would correspond to a scalar value greater than $10FFFF_{16}$ is illegal.

Because every Unicode coded character sequence maps to a unique sequence of code values in a given UTF, a reverse mapping can be derived. Thus every UTF supports *lossless round-trip transcoding*: mapping from any Unicode coded character sequence S to a sequence of code values and back will produce S again. To ensure that round-trip transcoding is possible, a UTF mapping *must also* map invalid Unicode scalar values to unique code value sequences. These invalid scalar values include $FFFE_{16}$, $FFFF_{16}$, and unpaired surrogates.

D30 For a given UTF, a code value sequence that cannot be produced from any sequence of Unicode scalar values is called an *ill-formed code value sequence*.

D31 For a given UTF, a code value sequence that cannot be mapped back to a sequence of Unicode scalar values is called an *illegal code value sequence*.

- For example, in UTF-8 every code value of the form $110xxxxx_2$ *must* be followed with a code value of the form $10xxxxxx_2$. A sequence such as $110xxxxx_2$ $0xxxxxx_2$ is illegal and must never be generated. When faced with this illegal code value sequence while transforming or interpreting, a UTF-8 conformant process must treat the first code value $110xxxxx_2$ as an illegal termination error—for example, by signaling an error, filtering the code value out, or representing the code value with a marker such as U+FFFD REPLACEMENT CHARACTER. In the latter two cases, it will continue processing at the second code value $0xxxxxx_2$.

D32 For a given UTF, an ill-formed code value sequence that is not illegal is called an *irregular code value sequence*.

- To make implementations simpler and faster, some transformation formats may allow irregular code value sequences without requiring error handling. For example, UTF-8 allows nonshortest code value sequences to be interpreted: a UTF-8 conformant process may map the code value sequence C0 80 (11000000_2 10000000_2) to the Unicode value U+0000, even though a UTF-8 conformant process shall *never* generate that code value sequence—it shall generate the sequence 00 (00000000_2) instead. A conformant process shall not use irregular code value sequences to encode out-of-band information.

D33 UTF-16BE is the Unicode Transformation Format that serializes a Unicode value as a sequence of two bytes, in big-endian format. An initial sequence corresponding to U+FEFF is interpreted as a ZERO WIDTH NO-BREAK SPACE.

- In UTF-16BE, <004D 0061 0072 006B> is serialized as <00 4D 00 61 00 72 00 6B>.

D34 UTF-16LE is the Unicode Transformation Format that serializes a Unicode value as a sequence of two bytes, in little-endian format. An initial sequence corresponding to U+FEFF is interpreted as a ZERO WIDTH NO-BREAK SPACE.

- In UTF-16LE, <004D 0061 0072 006B> is serialized as <4D 00 61 00 72 00 6B 00>.

D35 UTF-16 is the Unicode Transformation Format that serializes a Unicode value as a sequence of two bytes, in either big-endian or little-endian format. An initial byte sequence corresponding to U+FEFF is interpreted as a *byte order mark*: it is used to distinguish between the two byte orders. The *byte order mark* is not considered part of the content of the text. A serialization of Unicode values into UTF-16 may or may not begin with a *byte order mark*.

- In UTF-16, <004D 0061 0072 006B> is serialized as <FF FE 4D 00 61 00 72 00 6B 00>, <FE FF 00 4D 00 61 00 72 00 6B>, or <00 4D 00 61 00 72 00 6B>.
- The term *UTF-16* can be used ambiguously. When referring to the encoding of Unicode in memory, there is no associated byte orientation and a BOM is never used. When referring to a serialization of Unicode into bytes, it may have a BOM and can have either byte orientation.

D36 UTF-8 is the Unicode Transformation Format that serializes a Unicode scalar value as a sequence of one to four bytes, as specified in *Table 3-1*.

- In UTF-8, <004D 0061 0072 006B> is serialized as <4D 61 72 6B>.

Table 3-1 specifies the bit distribution from a Unicode character (or surrogate pair) into the one- to four-byte values of the corresponding UTF-8 sequence. Note that the four-byte form for surrogate pairs involves an addition of 10000_{16}, to account for the starting offset to the encoded values referenced by surrogates. The definition of UTF-8 in Amendment 2 to ISO/IEC 10646 also allows for the use of five- and six-byte sequences to encode characters that are outside the range of the Unicode character set; those five- and six-byte sequences are illegal for the use of UTF-8 as a transformation of Unicode characters.

Table 3-1. UTF-8 Bit Distribution

Scalar Value	UTF-16	1st Byte	2nd Byte	3rd Byte	4th Byte
000000000xxxxxxx	000000000xxxxxxx	0xxxxxxx			
00000yyyyyxxxxxx	00000yyyyyxxxxxx	110yyyyy	10xxxxxx		
zzzzyyyyyyxxxxxx	zzzzyyyyyyxxxxxx	1110zzzz	10yyyyyy	10xxxxxx	
uuuuuzzzzyyyyyyxxxxxx	110110wwwwzzzzyy+ 110111yyyyxxxxxx	11110uuu[a]	10uuzzzz	10yyyyyy	10xxxxxx

a. Where uuuuu = wwww + 1 (to account for addition of 10000_{16} as in *Section 3.7, Surrogates*).

When converting a Unicode scalar value to UTF-8, the shortest form that can represent those values shall be used. This practice preserves uniqueness of encoding. For example, the Unicode binary value <0000000000000001> is encoded as <00000001>, not as <11000000 10000001>. The latter is an example of an irregular UTF-8 byte sequence. *Irregular UTF-8 sequences shall not be used for encoding any other information.*

When converting from UTF-8 to a Unicode scalar value, implementations do not need to check that the shortest encoding is being used. This simplifies the conversion algorithm.

3.9 Special Character Properties

The behavior of most characters does not require special attention in this standard. However, the following characters exhibit special behavior, as described in *Chapter 13, Special Areas and Format Characters*, and in *Chapter 4, Character Properties*.

- Line boundary control

```
0009      HORIZONTAL TAB
000A      LINE FEED
000C      FORM FEED
000D      CARRIAGE RETURN
0020      SPACE
00A0      NO-BREAK SPACE
0F0B      TIBETAN MARK INTERSYLLABIC TSHEG
0F0C      TIBETAN MARK DELIMITER TSHEG BSTAR
2000      EN QUAD
2002      EN SPACE
2003      EM SPACE
2004      THREE-PER-EM SPACE
2005      FOUR-PER-EM SPACE
2006      SIX-PER-EM SPACE
2007      FIGURE SPACE
2008      PUNCTUATION SPACE
2009      THIN SPACE
200A      HAIR SPACE
200B      ZERO WIDTH SPACE
2011      NON-BREAKING HYPHEN
2028      LINE SEPARATOR
2029      PARAGRAPH SEPARATOR
202F      NARROW NO-BREAK SPACE
FEFF      ZERO WIDTH NO-BREAK SPACE
```

- Hyphenation control

```
002D      HYPHEN-MINUS
00AD      SOFT HYPHEN
058A      ARMENIAN HYPHEN
1806      MONGOLIAN TODO SOFT HYPHEN
2010      HYPHEN
2011      NON-BREAKING HYPHEN
2027      HYPHENATION POINT
```

- Fraction formatting

```
2044      FRACTION SLASH
```

- Special behavior with nonspacing marks

```
0020      SPACE
0069      LATIN SMALL LETTER I
006A      LATIN SMALL LETTER J
00A0      NO-BREAK SPACE
0131      LATIN SMALL LETTER DOTLESS I
```

- Double nonspacing marks

```
0360      COMBINING DOUBLE TILDE
0361      COMBINING DOUBLE INVERTED BREVE
0362      COMBINING DOUBLE RIGHTWARDS ARROW BELOW
```

- Joining

```
200C      ZERO WIDTH NON-JOINER
200D      ZERO WIDTH JOINER
```

- Bidirectional ordering

```
200E      LEFT-TO-RIGHT MARK
200F      RIGHT-TO-LEFT MARK
202A      LEFT-TO-RIGHT EMBEDDING
202B      RIGHT-TO-LEFT EMBEDDING
202C      POP DIRECTIONAL FORMATTING
202D      LEFT-TO-RIGHT OVERRIDE
```

```
202E      RIGHT-TO-LEFT OVERRIDE
```

- Alternate formatting

```
206A      INHIBIT SYMMETRIC SWAPPING
206B      ACTIVATE SYMMETRIC SWAPPING
206C      INHIBIT ARABIC FORM SHAPING
206D      ACTIVATE ARABIC FORM SHAPING
206E      NATIONAL DIGIT SHAPES
206F      NOMINAL DIGIT SHAPES
```

- Syriac abbreviation

```
070F      SYRIAC ABBREVIATION MARK
```

- Indic dead-character formation

```
094D      DEVANAGARI SIGN VIRAMA
09CD      BENGALI SIGN VIRAMA
0A4D      GURMUKHI SIGN VIRAMA
0ACD      GUJARATI SIGN VIRAMA
0B4D      ORIYA SIGN VIRAMA
0BCD      TAMIL SIGN VIRAMA
0C4D      TELUGU SIGN VIRAMA
0CCD      KANNADA SIGN VIRAMA
0D4D      MALAYALAM SIGN VIRAMA
0DCA      SINHALA SIGN AL-LAKUNA
0F84      TIBETAN SIGN HALANTA
1039      MYANMAR SIGN VIRAMA
17D2      KHMER SIGN COENG
```

- Mongolian variant selectors

```
180B      MONGOLIAN FREE VARIATION SELECTOR ONE
180C      MONGOLIAN FREE VARIATION SELECTOR TWO
180D      MONGOLIAN FREE VARIATION SELECTOR THREE
180E      MONGOLIAN VOWEL SEPARATOR
```

- Ideographic variation indication

```
303E      IDEOGRAPHIC VARIATION INDICATOR
```

- Ideographic description

```
2FF0      IDEOGRAPHIC DESCRIPTION CHARACTER LEFT TO
          RIGHT
2FF1      IDEOGRAPHIC DESCRIPTION CHARACTER ABOVE TO
          BELOW
2FF2      IDEOGRAPHIC DESCRIPTION CHARACTER LEFT TO
          MIDDLE AND RIGHT
2FF3      IDEOGRAPHIC DESCRIPTION CHARACTER ABOVE TO
          MIDDLE AND BELOW
2FF4      IDEOGRAPHIC DESCRIPTION CHARACTER FULL SUR-
          ROUND
2FF5      IDEOGRAPHIC DESCRIPTION CHARACTER SURROUND
          FROM ABOVE
2FF6      IDEOGRAPHIC DESCRIPTION CHARACTER SURROUND
          FROM BELOW
2FF7      IDEOGRAPHIC DESCRIPTION CHARACTER SURROUND
          FROM LEFT
2FF8      IDEOGRAPHIC DESCRIPTION CHARACTER SURROUND
          FROM UPPER LEFT
2FF9      IDEOGRAPHIC DESCRIPTION CHARACTER SURROUND
          FROM UPPER RIGHT
2FFA      IDEOGRAPHIC DESCRIPTION CHARACTER SURROUND
          FROM LOWER LEFT
2FFB      IDEOGRAPHIC DESCRIPTION CHARACTER OVERLAID
```

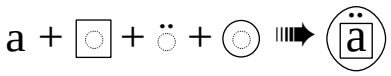
- Interlinear annotation
  FFF9 INTERLINEAR ANNOTATION ANCHOR
  FFFA INTERLINEAR ANNOTATION SEPARATOR
  FFFB INTERLINEAR ANNOTATION TERMINATOR
- Object replacement
  FFFC OBJECT REPLACEMENT CHARACTER
- Code conversion fallback
  FFFD REPLACEMENT CHARACTER
- Byte order signature
  FEFF ZERO WIDTH NO-BREAK SPACE

3.10 Canonical Ordering Behavior

The purpose of this section is to provide unambiguous interpretation of a combining character sequence. In the Unicode Standard, the order of characters in a combining character sequence is interpreted according to the following principles:

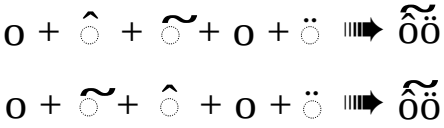
- In the Unicode Standard, all combining characters are encoded following the base characters to which they apply. Thus the Unicode sequence U+0061 "a" LATIN SMALL LETTER A + U+0308 "◌̈" COMBINING DIAERESIS + U+0075 "u" LATIN SMALL LETTER U is unambiguously interpreted (and displayed) as "äu", not "aü".
- Enclosing nonspacing marks surround all previous characters up to and including the base character (see *Figure 3-1*). They thus successively surround previous enclosing nonspacing marks.

Figure 3-1. Enclosing Marks



- Double diacritics always bind more loosely than other nonspacing marks. When rendering, the double diacritic will float above other diacritics, excluding enclosing diacritics (see *Figure 3-2*).

Figure 3-2. Positioning of Double Diacritics



- Combining marks with the same combining class are generally positioned graphically outward from the base character they modify. Some specific nonspacing marks override the default stacking behavior by being positioned side-by-side rather than stacking or by ligaturing with an adjacent nonspacing mark. When positioned side-by-side, the order of codes is reflected by positioning in the dominant order of the script with which they are used.

- If combining characters have different combining classes—for example, when one nonspacing mark is above a base character form and another is below it—then no distinction of graphic form or semantic will result.

The following subsections formalize these principles in terms of a normative list of combining classes and an algorithmic statement of how to use those combining classes to unambiguously interpret a combining character sequence.

Combining Classes

The Unicode Standard treats sequences of nonspacing marks as equivalent if they do not typographically interact. The canonical ordering algorithm defines a method for determining which sequences interact and gives a canonical ordering of these sequences for use in equivalence comparisons.

D37 *Combining class:* a numeric value given to each combining Unicode character that determines with which other combining characters it typographically interacts.

- See *Section 4.2, Combining Classes—Normative*, for a list of the combining classes for Unicode characters.

Characters have the same class if they interact typographically, and different classes if they do not.

- Enclosing characters and spacing combining characters have the class of their base characters.
- The particular numeric value of the combining class does not have any special significance; the intent of providing the numeric values is *only* to distinguish the combining classes as being different, for use in equivalence comparisons.

Canonical Ordering

The canonical ordering of a decomposed character sequence results from a sorting process that acts on each sequence of combining characters according to their combining class. Characters with combining class zero never sort relative to other characters, so the amount of work in the algorithm depends on the number of non-class-zero characters in a row. An implementation of this algorithm will be extremely fast for typical text.

The algorithm described here represents a logical description of the process. Optimized algorithms can be used in implementations as long as they are equivalent—that is, as long as they produce the same result.

More explicitly, the canonical ordering of a decomposed character sequence D results from the following algorithm.

R1 ***For each character x in D, let p(x) be the combining class of x.***

R2 ***Whenever any pair (A, B) of adjacent characters in D is such that p(B) ≠ 0 & p(A) > p(B), exchange those characters.***

R3 ***Repeat step R2 until no exchanges can be made among any of the characters in D.***

Examples of this ordering appear in *Table 3-2*.

Table 3-2. Sample Combining Classes

Combining Class	Abbreviation	Code	Unicode Name
0	a	0061	LATIN SMALL LETTER A
220	underdot	0323	COMBINING DOT BELOW
230	diaeresis	0308	COMBINING DIAERESIS
230	breve	0306	COMBINING BREVE
0	a-underdot	1EA1	LATIN SMALL LETTER A WITH DOT BELOW
0	a-diaeresis	00E4	LATIN SMALL LETTER A WITH DIAERESIS
0	a-breve	0103	LATIN SMALL LETTER A WITH BREVE

a + underdot + diaeresis ≡ *a + underdot + diaeresis*

a + diaeresis + underdot ≡ *a + underdot + diaeresis*

Because *underdot* has a lower combining class than *diaeresis*, the algorithm will return the *a*, then the *underdot*, then the *diaeresis*. However, because *diaeresis* and *breve* have the same combining class (because they interact typographically), they do not rearrange.

a + breve + diaeresis ≢ *a + diaeresis + breve*

a + diaeresis + breve ≢ *a + breve + diaeresis*

Applying the algorithm gives the results shown in *Table 3-3*.

Table 3-3. Canonical Ordering Results

Original	Decompose	Sort	Result
a-diaeresis + underdot	a + diaeresis + underdot	a + underdot + diaeresis	a + underdot + diaeresis
a + diaeresis + underdot		a + underdot + diaeresis	a + underdot + diaeresis
a + underdot + diaeresis			a + underdot + diaeresis
a-underdot + diaeresis	a + underdot + diaeresis		a + underdot + diaeresis
a-diaeresis + breve	a + diaeresis + breve		a + diaeresis + breve
a + diaeresis + breve			a + diaeresis + breve
a + breve + diaeresis			a + breve + diaeresis
a-breve + diaeresis	a + breve + diaeresis		a + breve + diaeresis

Use with Collation

When collation processes do not require correct sorting outside of a given domain, they are not required to invoke the canonical ordering algorithm for excluded characters. For example, a Greek collation process may not need to sort Cyrillic letters properly; in that case, it does not have to maximally decompose and reorder Cyrillic letters and may just choose to sort them according to Unicode order. For the complete treatment of collation, see Unicode Technical Report #10, "Unicode Collation Algorithm," on the CD-ROM or the up-to-date version on the Unicode Web site.

3.11 Conjoining Jamo Behavior

The Unicode Standard contains both a large set of precomposed modern Hangul syllables and a set of conjoining Hangul jamo, which can be used to encode archaic syllable blocks as well as modern syllable blocks. This section describes how to

- Determine the syllable boundaries in a sequence of conjoining jamo characters
- Compose jamo characters into Hangul syllables
- Determine the canonical decomposition of Hangul syllables
- Algorithmically determine the names of the Hangul syllable characters

(For more information, see the "Hangul Syllables" and "Hangul Jamo" subsections in *Section 10.4, Hangul*.)

The jamo characters can be classified into three sets of characters: *choseong* (leading consonants, or syllable-initial characters), *jungseong* (vowels, or syllable-peak characters), and *jongseong* (trailing consonants, or syllable-final characters). In the following discussion, these jamo are abbreviated as L (leading consonant), V (vowel), and T (trailing consonant); syllable breaks are shown by *middle dots* "·"; and non-jamo are shown by X.

Syllable Boundaries

In rendering, a sequence of jamos is displayed as a series of syllable blocks. The following rules specify how to divide up an arbitrary sequence of jamos (including nonstandard sequences) into these syllable blocks. In these rules, a *choseong filler* (L_f) is treated as a *choseong* character, and a *jungseong filler* (V_f) is treated as a *jungseong*.

Within any sequence of characters, a syllable break occurs between the pairs of characters shown in *Table 3-4*. All other sequences of Hangul jamos are considered to be part of the same syllable. Note that like other non-jamo characters, any combining mark between two conjoining jamos prevents the jamos from forming a syllable.

Table 3-4. Hangul Syllable Break Rules

Condition	Example
Any conjoining jamo and any non-jamo	L·X, V·X, T·X, X·L, X·V, X·T
A *jongseong* (trailing) and *choseong* (leading)	T·L
A *jungseong* (vowel) and a *choseong* (leading)	V·L
A *jongseong* (trailing) and *jungseong* (vowel)	T·V

Standard Syllables

A standard syllable block is composed of a sequence of *choseong* followed by a sequence of *jungseong* and optionally a sequence of *jongseong* (for example, $S = LV$ or LVT). A sequence of nonstandard syllable blocks can be transformed into a sequence of standard syllable blocks by inserting *choseong* fillers and *jungseong* fillers.

Examples. In *Table 3-5*, row (1) shows syllable breaks in a standard sequence, row (2) shows syllable breaks in a nonstandard sequence, and row (3) shows how the sequence in (2) could be transformed into standard form by inserting fillers into each syllable.

Table 3-5. Syllable Break Examples

No.	Sequence		Sequence with Syllable Breaks Marked
1	LVTLVLVLV L_f L_f VL V_f T	→	LVT · LV · LV · LV L_f · L V_f · L V T_f
2	LLTVLTLTVVLL	→	LLT · V · LT · LT · VV · LL
3	LLTVLTLTVVLL	→	LLV T_f · L V_f · LV T_f · LV T_f · L V_f V · LLV $_f$

Hangul Syllable Composition

The following algorithm describes how to take a sequence of canonically decomposed characters D and compose Hangul syllables. Hangul composition and decomposition are summarized here, but for a more complete description, implementers must consult Unicode Technical Report #15, "Unicode Normalization Forms," on the CD-ROM or the up-to-date version on the Unicode Web site. Note that, like other non-jamo characters, any combining mark between two conjoining jamos prevents the jamos from composing.

First, define the following constants:

```
SBase  = AC00₁₆
LBase  = 1100₁₆
VBase  = 1161₁₆
TBase  = 11A7₁₆
SCount = 11172
LCount = 19
VCount = 21
TCount = 28
NCount = VCount * TCount
```

1. Process D by composing the conjoining jamo wherever possible, according to the compatibility decomposition rules in *Chapter 14, Code Charts.* (Typical interchange of conjoining jamo will be in precomposed forms. In such cases, this step may not be necessary. Raw keyboard data, on the other hand, may be in the form of a compatibility decomposition.)

2. Let *i* represent the current position in the sequence D. Compute the following indices, which represent the ordinal number (zero-based) for each of the components of a syllable, and the index *j*, which represents the index of the last character in the syllable.

   ```
   LIndex = D[i] - LBase
   VIndex = D[i+1] - VBase
   TIndex = D[i+2] - TBase
   j      = i + 2
   ```

3. If either of the first two characters is out of bounds (LIndex < 0 OR LIndex >= LCount OR VIndex < 0 OR VIndex >= VCount), then increment *i*, return to step 2, and continue from there.

4. If the third character is out of bounds (TIndex <= 0 or TIndex >= TCount), then it is not part of the syllable. Reset the following:

   ```
   TIndex = 0
   j      = i + 1
   ```

5. Replace the characters D[*i*] through D[*j*] by the Hangul syllable S, and set *i* to be *j*+1.

   ```
   S      = (LIndex * VCount + VIndex) * TCount + TIndex + SBase.
   ```

Example. The first three characters are

U+1111	ᄑ	HANGUL CHOSEONG PHIEUPH
U+1171	ᅱ	HANGUL JUNGSEONG WI
U+11B6	ᆶ	HANGUL JONGSEONG RIEUL-HIEUH

Compute the following indices,

```
LIndex = 17
VIndex = 16
TIndex = 15
```

and replace the three characters by

```
S      = [(17 * 21) + 16] * 28 + 15 + SBase
       = D4DB16
       = 
```

Hangul Syllable Decomposition

The following describes the reverse mapping—how to take Hangul syllable S and derive the canonical decomposition D. This normative mapping for these characters is equivalent to the canonical mapping in the character charts for other characters.

1. Compute the index of the syllable:
   ```
   SIndex = S - SBase
   ```
2. If SIndex is in the range (0 <= SIndex < SCount), then compute the components as follows:
   ```
   L      = LBase + SIndex / NCount
   V      = VBase + (SIndex % NCount) / TCount
   T      = TBase + SIndex % TCount
   ```
   The operators "/" and "%" are as defined in *Section 0.2, Notational Conventions*.
3. If T = TBase, then there is no trailing character, so replace S by the sequence <L, V>. Otherwise, there is a trailing character, so replace S by the sequence <L, V, T>.

Example

```
L      = LBase + 17
V      = VBase + 16
T      = TBase + 15
D4DB16 → 111116, 117116, 11B616
```

Hangul Syllable Names

The character names for Hangul syllables are derived from the decomposition by starting with the string HANGUL SYLLABLE, and appending the short name of each decomposition component in order. (See *Section 4.4, Jamo Short Names—Normative*.) For example, for U+D4DB, derive the decomposition, as shown in the preceding example. It produces the following three-character sequence:

```
U+1111 HANGUL CHOSEONG PHIEUPH
U+1171 HANGUL JUNGSEONG WI
U+11B6 HANGUL JONGSEONG RIEUL-HIEUH
```

The character name for U+D4DB is then generated as HANGUL SYLLABLE PWILH. This character name is a normative property of the character.

3.12 Bidirectional Behavior

The Unicode Standard prescribes a *memory* representation order known as logical order. When text is presented in horizontal lines, most scripts display characters from left to right. However, in several scripts (such as Arabic or Hebrew), the natural ordering of horizontal text in display is from right to left. If all of the text has the same horizontal direction, then the ordering of the display text is unambiguous. However, when bidirectional text (a mixture of left-to-right and right-to-left horizontal text) is present, some ambiguities can arise in determining the ordering of the displayed characters.

This section describes the algorithm used to determine the directionality for bidirectional Unicode text. The algorithm extends the implicit model currently employed by a number of existing implementations and adds explicit format codes for special circumstances. In most cases, there is no need to include additional information with the text to obtain correct display ordering.

In the case of bidirectional text, there are circumstances where an implicit bidirectional ordering will not suffice to produce comprehensible text. To deal with these cases, a minimal set of directional formatting codes is defined to control the ordering of characters when rendered. This allows for exact control of the display ordering for legible interchange and also ensures that plain text used for simple items like file names or labels can always be correctly ordered for display.

The directional formatting codes are used *only* to influence the display ordering of text. In all other respects, they should be ignored—they have no effect on the comparison of text, word breaks, parsing, or numeric analysis.

When working with bidirectional text, the characters are still interpreted in logical order—only the display is affected. The display ordering of bidirectional text depends upon the directional properties of the characters in the text.

Directional Formatting Codes

Two types of explicit codes are used to modify the standard implicit Unicode bidirectional algorithm. In addition, there are implicit ordering codes, the *right-to-left* and *left-to-right* marks. All of these codes are limited to the current paragraph; thus their effects are terminated by a *paragraph separator*. The directional types left-to-right and right-to-left are called *strong types*, and characters of those types are called strong directional characters. The directional types associated with numbers are called *weak types*, and characters of those types are called weak directional characters.

Although the term *embedding* is used for some explicit codes, the text within the scope of the codes is not independent of the surrounding text. Characters within an embedding can affect the ordering of characters outside the embedding, and vice versa. The algorithm is designed so that the use of explicit codes can be equivalently represented by out-of-line information, such as stylesheet information. However, any alternative representation will be defined by reference to the behavior of the explicit codes in this algorithm.

Explicit Directional Embedding. The following codes signal that a piece of text is to be treated as embedded. For example, an English quotation in the middle of an Arabic sentence could be marked as being embedded left-to-right text. If a Hebrew phrase occurred in the middle of the English quotation, then that phrase could be marked as being embedded right-to-left. These codes allow for nested embeddings.

RLE	Right-to-Left Embedding	Treat the following text as embedded right-to-left.
LRE	Left-to-Right Embedding	Treat the following text as embedded left-to-right.

The precise meaning of these codes will be made clear in the discussion of the algorithm. The effect of right-left line direction, for example, can be accomplished by simply embedding the text with RLE...PDF.

Explicit Directional Overrides. The following codes allow the bidirectional character types to be overridden when required for special cases, such as for part numbers. These codes allow for nested directional overrides.

RLO	Right-to-Left Override	Force following characters to be treated as strong right-to-left characters.
LRO	Left-to-Right Override	Force following characters to be treated as strong left-to-right characters.

The precise meanings of these codes will be made clear in the discussion of the algorithm. The right-to-left override, for example, can be used to force a part number made of mixed English, digits, and Hebrew letters to be written from right to left.

Terminating Explicit Directional Code. The following code terminates the effects of the last explicit code (either embedding or override) and restores the bidirectional state to what it was before that code was encountered.

PDF	Pop Directional Format	Restore the bidirectional state to what it was before the last LRE, RLE, RLO, or LRO.

Implicit Directional Marks. These characters are very lightweight codes. They act exactly like right-to-left or left-to-right characters, except that they are not displayed and do not have any other semantic effect. Their use is generally more convenient than the explicit embeddings or overrides because their scope is much more local.

RLM	Right-to-Left Mark	Right-to-left zero-width character
LRM	Left-to-Right Mark	Left-to-right zero-width character

There is no special mention of the implicit directional marks in the following algorithm. That omission occurs because their effect on bidirectional ordering is exactly the same as a corresponding strong directional character; the only difference is that they do not appear in the display.

Basic Display Algorithm

The bidirectional algorithm takes a stream of text as input and proceeds in three main phases:

- Separation of the input text into paragraphs. The rest of the algorithm affects only the text between paragraph separators.
- Resolution of the embedding levels of the text. In this phase, the directional character types, plus the explicit format codes, are used to produce resolved embedding levels.
- Reordering the text for display on a line-by-line basis using the resolved embedding levels, once the text has been broken into lines.

The algorithm reorders text only within a paragraph; characters in one paragraph have no effect on characters in a different paragraph. Paragraphs are divided by U+2029 PARAGRAPH SEPARATOR or the appropriate Newline Function (see *Section 4.3, Directionality—Normative* and Unicode Technical Report #13, "Unicode Newline Guidelines," found on the CD-ROM or the up-to-date version on the Unicode Web site for information on the handling of CR, LF, and CRLF). Paragraphs may also be determined by higher-level protocols; for example, the text in two different cells of a table will be in different paragraphs.

Combining characters are always attached to the preceding base character in the memory representation. Even after reordering for display and performing character shaping, the glyph representing a combining character will be attached to the glyph representing its base character in memory. Depending on the line orientation and the placement direction of base letterform glyphs, it may, for example, become attached to the glyph on the left, or on the right, or above.

In the following subsections, the normative definitions and rules are distinguished by the numbering given in *Table 3-6*.

Table 3-6. Normative Definitions and Rules

Numbering	Section
BDn	Definitions
Pn	Paragraph levels
Xn	Explicit levels and directions
Wn	Weak types
Nn	Neutral types
In	Implicit levels
Ln	Resolved levels

Definitions

BD1 The *bidirectional character types* are values assigned to each Unicode character, including unassigned characters.

BD2 *Embedding levels* are numbers that indicate how deeply the text is nested, and the default direction of text on that level. The minimum embedding level of text is zero, and the maximum explicit depth is level 61.

- Embedding levels are explicitly set by both override format codes and embedding format codes; higher numbers mean the text is more deeply nested. The reason for having a limitation is to provide a precise stack limit for implementations to guarantee the same results. Sixty-one levels is far more than sufficient for ordering, even with mechanically generated formatting; the display becomes rather muddied with more than a small number of embeddings.

BD3 The default direction of the current embedding level (for a character in question) is called the *embedding direction*. It is L if the embedding level is even, and R if the embedding level is odd.

- For example, in a particular piece of text, level 0 is plain English text, level 1 is plain Arabic text, possibly embedded within English level 0 text. Level 2 is English text, possibly embedded within Arabic level 1 text, and so on. Unless their direction is overridden, English text and numbers will always be an even level; Arabic text (excluding numbers) will always be an odd level. The exact meaning of the embedding level will become clear when the reordering algorithm is discussed, but this section provides an example of how the algorithm works.

BD4 The *paragraph embedding level* is the embedding level that determines the default bidirectional orientation of the text in that paragraph.

BD5 The direction of the paragraph embedding level is called the *paragraph direction*.

- In some contexts, the paragraph direction is also known as the *base direction*.

BD6 The *directional override status* determines whether the bidirectional type of characters is reset with explicit directional controls. This status has three states, shown in *Table 3-7*.

BD7 A *level run* is a maximal substring of characters that have the same embedding level. It is maximal in that no character immediately before or after the substring has the same level.

Table 3-7. Directional Override Status

Status	Interpretation
Neutral	No override is currently active
Right-to-left	Characters are to be reset to R
Left-to-right	Characters are to be reset to L

Example. In this and the following examples, case is used to indicate different implicit character types for those unfamiliar with right-to-left letters. Uppercase letters stand for right-to-left characters (such as Arabic or Hebrew), whereas lowercase letters stand for left-to-right characters (such as English or Russian).

Memory:	`car is THE CAR in arabic`
Character types:	`LLL-LL-RRR-RRR-LL-LLLLLL`
Resolved levels:	`000000011111110000000000`

Notice that the neutral character (space) between THE and CAR gets the level of the surrounding characters. In this way, the implicit directional marks have an effect. By inserting appropriate directional marks around neutral characters, the level of the neutral characters can be changed.

Bidirectional Character Types. The normative bidirectional character types for each character are specified in the Unicode Character Database and summarized in *Table 3-8*.

Table 3-8. Bidirectional Character Types

Category	Type	Description	Scope
Strong	L	Left-to-Right	LRM, most alphabetic, syllabic, Han ideographic characters, digits that are neither European nor Arabic, all unassigned characters except in the ranges (0590–05FF, FB1D–FB4F) and (0600–07BF, FB50–FDFF, FE70–FEFF)
	LRE	Left-to-Right Embedding	LRE
	LRO	Left-to-Right Override	LRO
	R	Right-to-Left	RLM, Hebrew alphabet, most punctuation specific to that script, all unassigned characters in the ranges (0590–05FF, FB1D–FB4F)
	AL	Right-to-Left Arabic	Arabic, Thaana, and Syriac alphabets, most punctuation specific to those scripts, all unassigned characters in the ranges (0600–07BF, FB50–FDFF, FE70–FEFF)
	RLE	Right-to-Left Embedding	RLE
	RLO	Right-to-Left Override	RLO

Table 3-8. Bidirectional Character Types (Continued)

Category	Type	Description	Scope
Weak	PDF	Pop Directional Format	PDF
	EN	European Number	European digits, Eastern Arabic-Indic digits, ...
	ES	European Number Separator	Solidus (slash)
	ET	European Number Terminator	Plus sign, minus sign, degree, currency symbols, ...
	AN	Arabic Number	Arabic-Indic digits, Arabic decimal and thousands separators, ...
	CS	Common Number Separator	Colon, comma, full stop (period), NO-BREAK SPACE, ...
	NSM	Nonspacing Mark	Characters marked Mn (nonspacing mark) and Me (enclosing mark) in the Unicode Character Database
	BN	Boundary Neutral	Formatting and control characters, other than those explicitly given types above (to be ignored in processing bidirectional text)
Neutral	B	Paragraph Separator	Paragraph separator, appropriate newline functions, higher-protocol paragraph determination
	S	Segment Separator	Tab
	WS	Whitespace	Space, figure space, line separator, form feed, general punctuation spaces, ...
	ON	Other Neutrals	All other characters, including OBJECT REPLACEMENT CHARACTER

- The term European digits is used to refer to decimal forms common in Europe and elsewhere, and the term Arabic-Indic digits refers to the native Arabic forms. (See *Section 8.2, Arabic*, for more details on naming digits.)
- Unassigned characters are given strong types in the algorithm. This convention is an explicit exception to the general Unicode conformance requirements with respect to unassigned characters. As characters become assigned in the future, these bidirectional types may change.
- Private-use characters can be assigned different values by a conformant implementation.
- For the purpose of the bidirectional algorithm, inline objects (such as graphics) are treated as if they are an OBJECT REPLACEMENT CHARACTER (U+FFFC).

Table 3-9 lists additional abbreviations used in the examples and internal character types used in the algorithm.

Table 3-9. Abbreviations for Examples and Internal Types

Symbol	Description
N	Neutral or separator (B, S, WS, ON).
e	The text ordering type (L or R) that matches the embedding level direction (even or odd) for the current run. Cf. X10 below.
sor	The text ordering type (L or R) assigned to the position before a level run.
eor	The text ordering type (L or R) assigned to the position after a level run.

Resolving Embedding Levels

The body of the bidirectional algorithm uses character types and explicit codes to produce a list of resolved levels. This resolution process consists of five steps: (1) determining the paragraph level; (2) determining explicit embedding levels and directions; (3) resolving weak types; (4) resolving neutral types; and (5) resolving implicit embedding levels.

Paragraph Level. The bidirectional algorithm applies to paragraphs.

P1. Split the text into separate paragraphs. A paragraph separator is kept with the previous paragraph. Within each paragraph, apply all other rules of this algorithm.

P2. In each paragraph, find the first character that is a strong directional type (L, AL, R).

Because paragraph separators delimit text in this algorithm, they will generally be the first strong character after a paragraph separator or at the very beginning of the text.

P3. If a character is found in rule P2 and it is of type AL or R, then set the paragraph embedding level to 1; otherwise, set it to zero.

Note that when a higher-level protocol specifies the paragraph level, it is not necessary to apply rules P2 and P3.

Explicit Levels and Directions. All explicit embedding levels are determined from the embedding and override codes, by applying the explicit level rules X1 through X9. These rules are applied as part of the same logical pass over the input.

Explicit Embeddings. An explicit embedding code sets the level of the text, but does not change the directional character type of affected characters.

X1. Begin by setting the current embedding level to the paragraph embedding level. Set the directional override status to neutral. Process each character iteratively, applying rules X2 through X9. Only embedding levels from 0 to 61 are valid in this phase.

In the resolution of levels in rules I1 and I2, the maximum embedding level of 62 can be reached.

X2. With each RLE, compute the least greater odd *embedding level.*

a. If this new level would be valid, then this embedding code is valid. Remember (push) the current embedding level and override status. Reset the current level to this new level, and reset the override status to neutral.

b. If the new level would not be valid, then this code is invalid. Don't change the current level or override status.

For example, level 0 → 1; levels 1, 2 → 3; levels 3, 4 → 5; ...59, 60 → 61; above 60, no change (don't change levels with RLE if the new level would be invalid).

X3. With each LRE, compute the least greater even *embedding level.*

a. If this new level would be valid, then this embedding code is valid. Remember (push) the current embedding level and override status. Reset the current level to this new level, and reset the override status to neutral.

b. If the new level would not be valid, then this code is invalid. Don't change the current level or override status.

For example, levels 0, 1 → 2; levels 2, 3 → 4; levels 4, 5 → 6; ...58, 59 → 60; above 59, no change (don't change levels with LRE if the new level would be invalid).

Explicit Overrides. An explicit directional override sets the embedding level in the same way that the explicit embedding codes do, but also changes the directional character type of affected characters to the override direction.

X4. With each RLO, compute the least greater odd *embedding level.*

a. If this new level would be valid, then this embedding code is valid. Remember (push) the current embedding level and override status. Reset the current level to this new level, and reset the override status to right-to-left.

b. If the new level would not be valid, then this code is invalid. Don't change the current level or override status.

X5. With each LRO, compute the least greater even *embedding level.*

a. If this new level would be valid, then this embedding code is valid. Remember (push) the current embedding level and override status. Reset the current level to this new level, and reset the override status to left-to-right.

b. If the new level would not be valid, then this code is invalid. Don't change the current level or override status.

X6. For all types besides RLE, LRE, RLO, LRO, and PDF:

a. Set the level of the current character to the current embedding level.

b. Whenever the directional override status is not neutral, reset the current character type to the directional override status.

If the directional override status is neutral, then characters retain their normal types: Arabic characters stay AL, Latin characters stay L, neutrals stay N, and so on. If the directional override status is R, then characters become R. If the directional override status is L, then characters become L.

Terminating Embeddings and Overrides. A single code terminates the scope of the current explicit code, whether an embedding or a directional override. All codes and pushed states are completely popped at the end of paragraphs.

X7. With each PDF, determine the matching embedding or override code. If a valid matching code was found, restore (pop) the last remembered (pushed) embedding level and directional override.

X8. All explicit directional embeddings and overrides are completely terminated at the end of each paragraph. Paragraph separators are not *included in the embedding.*

X9. Remove all RLE, LRE, RLO, LRO, PDF, and BN codes.

- Note that an implementation does not have to actually remove the codes, it just has to behave as though the codes were not present for the remainder of the algorithm. Conformance does not require any particular placement of these codes as long as all other characters are ordered correctly.

See "Implementation Notes" later in this section for information on implementing the algorithm without removing the formatting codes.

X10. The remaining rules are applied to each run of characters at the same level. For each run, determine the start-of-level-run (sor) *and end-of-level-run* (eor) *type, either L or R. This determination depends on the higher of the two levels on either side of the boundary (at the start or end of the paragraph, the level of the "other" run is the base embedding level). If the higher level is odd, the type is R; otherwise, it is L.*

For example:

Levels: `0 0 0 1 1 1 2`
Runs: ← `1` → ← `2` → `<3>`

Run 1 is at level 0, *sor* is L, *eo*r is R.

Run 2 is at level 1, *sor* is R, *eor* is L.

Run 3 is at level 2, *sor* is L, *eor* is L.

For two adjacent runs, the *eor* of the first run is the same as the *sor* of the second.

Resolving Weak Types. Weak types are now resolved one level run at a time. At level run boundaries where the type of the character on the other side of the boundary is required, the type assigned to *sor* or *eor* is used.

Nonspacing marks are now resolved based on the previous characters.

W1. Examine each nonspacing mark (NSM) in the level run, and change the type of the NSM to the type of the previous character. If the NSM is at the start of the level run, it will get the type of sor.

Assume in this example that *sor* is R:

```
AL  NSM NSM → AL  AL  AL
sor NSM     → sor R
```

The text is next parsed for numbers. This pass will change the directional types European Number Separator, European Number Terminator, and Common Number Separator to be European Number text, Arabic Number text, or Other Neutral text. The text to be scanned may have already had its type altered by directional overrides. If so, then it will not parse as numeric.

W2. Search backward from each instance of a European number until the first strong type (R, L, AL, or sor*) is found. If an AL is found, change the type of the European number to Arabic number.*

```
AL EN    → AL AN
AL N EN  → AL N AN
sor N EN → sor N EN
L N EN   → L N EN
R N EN   → R N EN
```

W3. Change all ALs to R.

W4. A single European separator between two European numbers changes to a European number. A single common separator between two numbers of the same type changes to that type:

```
EN ES EN → EN EN EN
EN CS EN → EN EN EN
AN CS AN → AN AN AN
```

W5. A sequence of European terminators adjacent to European numbers changes to all European numbers:

```
ET ET EN → EN EN EN
EN ET ET → EN EN EN
AN ET EN → AN EN EN
```

W6. Otherwise, separators and terminators change to Other Neutral:

```
AN ET    → AN ON
L  ES EN → L  ON EN
EN CS AN → EN ON AN
ET AN    → ON AN
```

W7. Search backward from each instance of a European number until the first strong type (R, L, or sor*) is found. If an L is found, then change the type of the European number to L.*

```
L  N EN  → L  N  L
R  N EN  → R  N  EN
```

Resolving Neutral Types. Neutral types are now resolved one level run at a time. At level run boundaries where the type of the character on the other side of the boundary is required, the type assigned to *sor* or *eor* is used.

The next phase resolves the direction of the neutrals. The results of this phase are that all neutrals become either R or L. Generally, neutrals take on the direction of the surrounding text. In case of a conflict, they take on the embedding direction.

*N1. A sequence of neutrals takes the direction of the surrounding strong text if the text on both sides has the same direction. European and Arabic numbers are treated as though they were R. Start-of-level-run (*sor*) and end-of-level-run (*eor*) are used at level run boundaries.*

```
R  N  R  → R  R  R
L  N  L  → L  L  L
R  N  AN → R  R  AN
AN N  R  → AN R  R
R  N  EN → R  R  EN
EN N  R  → EN R  R
EN N  AN → EN R  AN
```

- Any European numbers after an L have already been turned into L by this time, so only the European numbers after R are treated as R.

N2. Any remaining neutrals take the embedding direction.

```
N → e
```

Assume in this example that *eor* is L, and *sor* is R:

```
L   N eor  → L   L eor
R   N eor  → R   e eor
sor N L    → sor e L
sor N R    → sor R R
```

Examples. A list of numbers separated by neutrals and embedded in a directional run will come out in the run's order.

Storage: `he said “THE VALUES ARE 123, 456, 789, OK”.`

Display: `he said “KO ,789 ,456 ,123 ERA SEULAV EHT”.`

In this case, both the comma and the space between the numbers take on the direction of the surrounding text (uppercase = right-to-left), ignoring the numbers. The commas are not considered part of the number because they are not surrounded on both sides. If an adjacent left-to-right sequence is present, then European numbers will adopt that direction.

Storage: `he said “IT IS A bmw 500, OK.”`

Display: `he said “.KO ,bmw 500 A SI TI”`

Resolving Implicit Levels. In the final phase, the embedding level of text may be increased, based upon the resolved character type. Right-to-left text will always end up with an odd level, and left-to-right and numeric text will always end up with an even level. In addition, numeric text will always end up with a higher level than the paragraph level. (Note that it is

possible for text to end up at levels higher than 61 as a result of this process.) This situation results in the following rules:

I1. For all characters with an even (left-to-right) embedding direction, those of type R go up one level and those of type AN or EN go up two levels.

I2. For all characters with an odd (right-to-left) embedding direction, those of type L, EN, or AN go up one level.

Table 3-10 summarizes the results of the implicit algorithm.

Table 3-10. Resolving Implicit Levels

Type	Embedding Level	
	Even	**Odd**
L	EL	EL+1
R	EL+1	EL
AN	EL+2	EL+1
EN	EL+2	EL+1

Reordering Resolved Levels

The following algorithm describes the logical process of finding the correct display order. As noted earlier, this logical process is not necessarily the actual implementation, which may diverge for efficiency as long as it produces the same results. As opposed to resolution phases, this algorithm acts on a per-line basis, *and is applied* after *any line wrapping is applied to the paragraph.*

The process of breaking a paragraph into one or more lines that fit within particular bounds is outside the scope of the bidirectional algorithm. Where character shaping is involved, it can be somewhat more complicated (see *Section 8.2, Arabic*). Logically, there are the following steps:

- The levels of the text are determined according to the bidirectional algorithm.
- The characters are shaped into glyphs according to their context (*taking the embedding levels into account for mirroring!*).
- The accumulated widths of those glyphs (*in logical order*) are used to determine line breaks.
- For each line, rules L1–L4 are used to reorder the characters on that line.
- The glyphs corresponding to the characters on the line are displayed in that order.

L1. On each line, reset the embedding level of the following characters to the paragraph embedding level:

1. *segment separators,*
2. *paragraph separators,*
3. *any sequence of whitespace characters preceding a segment separator or paragraph separator, and*
4. *any sequence of whitespace characters at the end of the line.*

- The types of characters used here are the *original* types, not those modified by the previous phase.

- Because a *paragraph separator* breaks lines, there will be at most one per line, at the end of that line.

In combination with the following rule, this requirement means that trailing whitespace will appear at the visual end of the line (in the paragraph direction). Tabulation will always have a consistent direction within a paragraph.

L2. From the highest level found in the text to the lowest odd level on each line, reverse any contiguous sequence of characters that is at that level or higher.

This process reverses a progressively larger series of substrings. The following four examples illustrate this rule.

Memory:	`car means CAR.`
Resolved levels:	`00000000001110`
Reverse level 1:	`car means RAC.`
Memory:	`car MEANS CAR.`
Resolved levels:	`22211111111111`
Reverse level 2:	`rac MEANS CAR.`
Reverse levels 1, 2:	`.RAC SNAEM car`
Memory:	`he said “car MEANS CAR.”`
Resolved levels:	`000000000222111111111100`
Reverse level 2:	`he said “rac MEANS CAR.”`
Reverse levels 1, 2:	`he said “RAC SNAEM car.”`
Memory:	`DID YOU SAY ‘he said “car MEANS CAR”’?`
Resolved levels:	`11111111111112222222224443333333333211`
Reverse level 4:	`DID YOU SAY ‘he said “rac MEANS CAR”’?`
Reverse levels 3, 4:	`DID YOU SAY ‘he said “RAC SNAEM car”’?`
Reverse levels 2–4:	`DID YOU SAY ’”rac MEANS CAR“ dias eh‘?`
Reverse levels 1–4:	`?‘he said “RAC SNAEM car”’ YAS UOY DID`

L3. Combining marks applied to a right-to-left base character will at this point precede their base character. If the rendering engine expects them to follow the base characters in the final display process, then the ordering of the marks and the base character must be reversed.

Many font designers provide default metrics for combining marks that support rendering by simple overhang. Because of the reordering for right-to-left characters, it is common practice to make the glyphs for most combining characters overhang to the left (thereby assuming the characters will be applied to left-to-right base characters) and make the glyphs for combining characters in right-to-left scripts overhang to the right (thereby assuming that the characters will be applied to right-to-left base characters). With such fonts, the display ordering of the marks and base glyphs may need to be adjusted when combining marks are applied to “unmatching” base characters. See *Section 5.14, Rendering Nonspacing Marks*, for more information.

L4. A character that possesses the mirrored property as specified by Section 4.7, Mirrored—Normative, must be depicted by a mirrored glyph if the resolved directionality of that character is R.

For example, U+0028 LEFT PARENTHESIS—which is interpreted in the Unicode Standard as an opening parenthesis—appears as “(” when its resolved level is even, and as the mirrored glyph “)” when its resolved level is odd.

Bidirectional Conformance

The bidirectional algorithm specifies part of the intrinsic semantics of right-to-left characters. In the absence of a higher-level protocol that specifically supersedes the interpretation of directionality, systems that interpret these characters must achieve results identical to the implicit bidirectional algorithm when rendering.

Boundary Neutrals. The goal in marking a format or control character as BN is to ensure that it has no effect on the rest of the algorithm. Because the precise ordering of format characters with respect to others is not required for conformance, implementations are free to handle them in different ways for efficiency as long as the ordering of the other characters is preserved.

Explicit Formatting Codes. As with any Unicode characters, systems do not have to support any particular explicit directional formatting code (although it is not generally useful to include a terminating code without including the initiator). Generally, conforming systems will fall into three classes:

- *No bidirectional formatting.* This choice implies that the system does not visually interpret characters from right-to-left scripts.
- *Implicit bidirectionality.* The implicit bidirectional algorithm and the directional marks RLM and LRM are supported.
- *Full bidirectionality.* The implicit bidirectional algorithm, the implicit directional marks, and the explicit directional embedding codes are supported: RLM, LRM, LRE, RLE, LRO, RLO, PDF.

Higher-Level Protocols. The following are permissible ways for systems to apply higher-level protocols to the ordering of bidirectional text:

- *Override the paragraph embedding level.* A higher-level protocol should provide for overriding the paragraph embedding level, such as on a table cell, paragraph, document, or system level.
- *Override the number handling to use information provided by a broader context.* For example, information from other paragraphs in a document could be used to conclude that the document was fundamentally Arabic and that EN should generally be converted to AN.
- *Replace, supplement, or override the directional overrides or embedding codes.* This task is accomplished by providing information via additional stylesheet or markup information about the embedding level or character direction. The interpretation of such information must always be defined by reference to the behavior of the equivalent explicit codes as given in the algorithm.
- *Override the bidirectional character types assigned to control codes to match the interpretation of the control codes within the protocol.* (See also *Section 13.1, Control Codes.*)
- *Remap the number shapes to match those of another set.* For example, remap the Arabic number shapes to have the same appearance as the European numbers.

When text using a higher-level protocol is to be converted to Unicode plain text, formatting codes should be inserted to ensure that the order matches that of the higher-level protocol or that (as in the last example) the appropriate characters can be substituted.

Implementation Notes

Reference Implementations. Source code for two reference implementations of the bidirectional algorithm is provided as part of Unicode Technical Report #9, "The Bidirectional Algorithm," on the CD-ROM or the up-to-date version on the Unicode Web site. Implementers are encouraged to use this resource to test their implementations.

Retaining Format Codes. Some implementations may wish to retain the format codes when running the algorithm. This goal may be accomplished in the following manner:

- In rule X9, instead of removing the format codes, set all format codes to BN. For each sequence of BN codes, set the levels of the codes to the level of the preceding non-BN code (or if the sequence is at the beginning of the paragraph, to the paragraph level).
- In rule W1, search backward from each NSM to the first character in the level run whose type is not BN, and set the NSM to its type.
- In rule W4, scan past BN types that are adjacent to ES or CS.
- In rule W5, change all appropriate sequences of ET and BN, not just ET.
- In rule W6, change all BN types to ON as well.
- In rule L1, include format codes and BN together with whitespace characters in the sequences whose levels are reset before a separator or line break.

Implementations that display visible representations of format characters will want to adjust this mechanism so as to position the format characters optimally for editing.

Vertical Text. In the case of vertical line orientation, the bidirectional algorithm is still used to determine the levels of the text. However, these levels are not used to reorder the text, as the characters are usually ordered uniformly from top to bottom. Instead, the levels are used to determine the rotation of the text. Sometimes vertical lines follow a vertical baseline in which each character is oriented as normal (with no rotation), with characters ordered from top to bottom whether they are Hebrew, numbers, or Latin. When setting text using the Arabic script in vertical lines, it is more common to employ a horizontal baseline that is rotated by 90 degrees counterclockwise so that the characters are ordered from top to bottom. Latin text and numbers may be rotated 90 degrees clockwise so that the characters become ordered from top to bottom as well.

The bidirectional algorithm also comes into play when some characters are ordered from bottom to top. For example, this situation arises with a mixture of Arabic and Latin glyphs when all of the glyphs are rotated uniformly 90 degrees clockwise. (The choice of whether text is to be presented horizontally or vertically, or whether text is to be rotated, is not specified by the Unicode Standard, and is left to higher-level protocols.)

Usage. Because of the implicit character types and the heuristics for resolving neutral and numeric directional behavior, the implicit bidirectional ordering will generally produce the correct display without any further work. However, problematic cases may occur when a right-to-left paragraph begins with left-to-right characters, or nested segments of different-direction text are present, or there are weak characters on directional boundaries. In these cases, embeddings or directional marks may be required to get the correct display. Part numbers may also require directional overrides.

The most common problematic case involves neutrals on the boundary of an embedded language. This issue can be addressed by setting the level of the embedded text correctly. For example, with all text at level 0, the following occurs:

Memory: `he said "I NEED WATER!", and expired.`

Display: `he said "RETAW DEEN I!", and expired.`

If the exclamation mark is to be part of the Arabic quotation, then the user can select the text *I NEED WATER!* and explicitly mark it as embedded Arabic, which produces the following result:

Memory: `he said "<RLE>I NEED WATER!<PDF>", and expired.`

Display: `he said "!RETAW DEEN I", and expired.`

A simpler method is to place a right directional mark (RLM) after the exclamation mark. Because the exclamation mark is no longer on a directional boundary, this produces the correct result.

Memory: `he said "I NEED WATER!<RLM>", and expired.`

Display: `he said "!RETAW DEEN I", and expired.`

This latter approach is preferred because it does not use the stateful format codes, which can easily get out of sync if not fully supported by editors and other string manipulation. The stateful format codes are generally needed only for more complex (and rare) cases such as double embeddings, as in the following:

Memory: `DID YOU SAY ‘<LRE>he said "I NEED WATER!<RLM>", and expired.<PDF>’?`

Display: `?‘he said "!RETAW DEEN I", and expired.’ YAS UOY DID`

Migrating from 2.0 to 3.0. In the Unicode Character Database for Version 3.0, new bidirectional character types are introduced to make the body of the algorithm depend only on the types of characters, and not on the character values. The changes from the Version 2.0 bidirectional types are listed in *Table 3-11*.

Table 3-11. New Bidirectional Types

Characters	New Bidirectional Type
All characters with General Category Me, Mn	NSM
All characters of type R in the Arabic ranges (0600..06FF, FB50..FDFF, FE70..FEFE) (Letters in the Thaana and Syriac ranges also have this value.)	AL
The explicit embedding characters: LRO, RLO, LRE, RLE, PDF	LRO, RLO, LRE, RLE, PDF, respectively
Formatting characters and controls (General Category Cf and Cc) that were of bidirectional type ON	BN
Zero Width Space	BN

Implementations that use older property tables can be adjusted to the modifications in the bidirectional algorithm by algorithmically remapping the characters listed in *Table 3-11* to the new types.

This PDF file is an excerpt from *The Unicode Standard, Version 3.0*, issued by the Unicode Consortium and published by Addison-Wesley. The material has been modified slightly for this online edition, however the PDF files have not been modified to reflect the corrections found on the Updates and Errata page (see http://www.unicode.org/unicode/uni2errata/UnicodeErrata.html). More recent versions of the Unicode standard exist (see http://www.unicode.org/unicode/standard/versions/).

Dai Kan-Wa Jiten used as the source of reference Kanji codes was written by Tetsuji Morohashi and published by Taishukan Shoten.

ISBN 0-201-61633-5

This book is set in Minion, designed by Rob Slimbach at Adobe Systems, Inc. It was typeset using FrameMaker 5.5 running under Windows NT. ASMUS, Inc. created custom software for chart layout. The Han radical-stroke index was typeset by Apple Computer, Inc. The following companies and organizations supplied fonts:

Apple Computer, Inc.
Atelier Fluxus Virus
Beijing Zhong Yi (Zheng Code) Electronics Company
DecoType, Inc.
IBM Corporation
Monotype Typography, Inc.
Microsoft Corporation
Peking University Founder Group Corporation
Production First Software

Additional fonts were supplied by individuals as listed in the *Acknowledgments*.

The publisher offers discounts on this book when ordered in quantity for special sales. For more information please contact:

Corporate, Government, and Special Sales
Addison Wesley Longman, Inc.
One Jacob Way
Reading, Massachusetts 01867

Visit A-W on the Web: http://www.awl.com/cseng/

First printing, January 2000.