

Proposed Update Unicode Standard Annex #9

UNICODE BIDIRECTIONAL ALGORITHM

Version	Unicode 6.2.1 (draft 3)
Editors	Mark Davis (markdavis@google.com)
Date	2012-10-26
This Version	http://www.unicode.org/reports/tr9/tr9-28.html
Previous Version	http://www.unicode.org/reports/tr9/tr9-27.html
Latest Version	http://www.unicode.org/reports/tr9/
Latest Proposed Update	http://www.unicode.org/reports/tr9/proposed.html
Revision	28

Summary

This annex describes specifications for the positioning of characters in text containing characters flowing from right to left, such as Arabic or Hebrew.

Status

*This is a **draft** document which may be updated, replaced, or superseded by other documents at any time. Publication does not imply endorsement by the Unicode Consortium. This is not a stable document; it is inappropriate to cite this document as other than a work in progress.*

A Unicode Standard Annex (UAX) forms an integral part of the Unicode Standard, but is published online as a separate document. The Unicode Standard may require conformance to normative content in a Unicode Standard Annex, if so specified in the Conformance chapter of that version of the Unicode Standard. The version number of a UAX document corresponds to the version of the Unicode Standard of which it forms a part.

Please submit corrigenda and other comments with the online reporting form [[Feedback](#)]. Related information that is useful in understanding this annex is found in

Unicode Standard Annex #41, “[Common References for Unicode Standard Annexes](#).”
For the latest version of the Unicode Standard, see [[Unicode](#)]. For a list of current Unicode Technical Reports, see [[Reports](#)]. For more information about versions of the Unicode Standard, see [[Versions](#)]. For any errata which may apply to this annex, see [[Errata](#)].

Contents

- 1 [Introduction](#)
 - 2 [Directional Formatting Codes](#)
 - 2.1 [Explicit Directional Embedding](#)
 - 2.2 [Explicit Directional Overrides](#)
 - 2.3 [Terminating Explicit Directional Embeddings and Overrides](#)
 - 2.4 [Explicit Directional Isolates](#)
 - 2.5 [Terminating Explicit Directional Isolates](#)
 - 2.6 [Implicit Directional Marks](#)
 - 3 [Basic Display Algorithm](#)
 - 3.1 [Definitions: \[BD1\]\(#\), \[BD2\]\(#\), \[BD3\]\(#\), \[BD4\]\(#\), \[BD5\]\(#\), \[BD6\]\(#\), \[BD7\]\(#\), \[BD8\]\(#\), \[BD9\]\(#\), \[BD10\]\(#\)](#)
 - 3.2 [Bidirectional Character Types](#)
 - 3.3 [Resolving Embedding Levels](#)
 - 3.3.1 [The Paragraph Level: \[P1\]\(#\), \[P2\]\(#\), \[P3\]\(#\)](#)
 - 3.3.2 [Explicit Levels and Directions: \[X1\]\(#\), \[X2\]\(#\), \[X3\]\(#\), \[X4\]\(#\), \[X5\]\(#\), \[X5a\]\(#\), \[X5b\]\(#\), \[X5c\]\(#\), \[X6\]\(#\), \[X6a\]\(#\), \[X7\]\(#\), \[X8\]\(#\), \[X9\]\(#\), \[X10\]\(#\)](#)
 - 3.3.3 [Resolving Weak Types: \[W1\]\(#\), \[W2\]\(#\), \[W3\]\(#\), \[W4\]\(#\), \[W5\]\(#\), \[W6\]\(#\), \[W7\]\(#\)](#)
 - 3.3.4 [Resolving Neutral Types: \[N0\]\(#\), \[N1\]\(#\), \[N2\]\(#\)](#)
 - 3.3.5 [Resolving Implicit Levels: \[I1\]\(#\), \[I2\]\(#\)](#)
 - 3.4 [Reordering Resolved Levels: \[L1\]\(#\), \[L2\]\(#\), \[L3\]\(#\), \[L4\]\(#\)](#)
 - 3.5 [Shaping](#)
 - 4 [Bidirectional Conformance](#)
 - 4.1 [Boundary Neutrals](#)
 - 4.2 [Explicit Formatting Codes](#)
 - 4.3 [Higher-Level Protocols: \[HL1\]\(#\), \[HL2\]\(#\), \[HL3\]\(#\), \[HL4\]\(#\), \[HL5\]\(#\), \[HL6\]\(#\)](#)
 - 4.4 [Bidi Conformance Testing](#)
 - 5 [Implementation Notes](#)
 - 5.1 [Reference Code](#)
 - 5.2 [Retaining Format Codes](#)
 - 5.3 [Joiners](#)
 - 5.4 [Vertical Text](#)
 - 5.5 [Usage](#)
 - 5.6 [Separating Punctuation Marks](#)
 - 5.7 [Migrating from 2.0 to 3.0](#)
 - 5.8 [Conversion to Plain Text](#)
 - 6 [Mirroring](#)
 - [Acknowledgments](#)
 - [References](#)
 - [Modifications](#)
-

1 Introduction

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, there are several scripts (such as Arabic or Hebrew) where the natural ordering of horizontal text in display is from right to left. If all of the text has a uniform horizontal direction, then the ordering of the display text is unambiguous.

However, because these right-to-left scripts use digits that are written from left to right, the text is actually *bidirectional*: a mixture of right-to-left *and* left-to-right text. In addition to digits, embedded words from English and other scripts are also written from left to right, also producing bidirectional text. Without a clear specification, ambiguities can arise in determining the ordering of the displayed characters when the horizontal direction of the text is not uniform.

This annex 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. Each character has an implicit *bidirectional type*. The bidirectional types left-to-right and right-to-left are called *strong types*, and characters of those types are called strong directional characters. The bidirectional types associated with numbers are called *weak types*, and characters of those types are called weak directional characters. The algorithm uses the implicit bidirectional types of the characters in a text to arrive at a reasonable display ordering for text. In most cases, there is no need to include additional information with the text to obtain correct display ordering.

However, in the case of bidirectional text, there are circumstances where an implicit bidirectional ordering is not sufficient 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 exact control of the display ordering for legible interchange and ensures that plain text used for simple items like filenames 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 or on 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 on the directional properties of the characters in the text. Note that there are important security issues connected with bidirectional text: for more information, see [\[UTR36\]](#).

2 Directional Formatting Codes

Three types of explicit codes are used to modify the standard implicit Unicode Bidirectional Algorithm (UBA). 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.

These controls all have the property *Bidi_Control*, and are divided into three ranges:

Implicit Bidi Controls

U+200E..U+200F LEFT-TO-RIGHT MARK..RIGHT-TO-LEFT MARK

Explicit Bidi Controls

U+202A..U+202E LEFT-TO-RIGHT EMBEDDING..RIGHT-TO-LEFT OVERRIDE

Explicit Bidi Controls for Isolates

Proposed U+2066..U+2069 LEFT-TO-RIGHT ISOLATE..POP DIRECTIONAL ISOLATE



Although specific codepoints have been slated for the explicit bidi controls for isolates, they will not be formally published until Unicode 7.0. Nevertheless, Unicode 6.2.1 already defines the corresponding bidirectional character types, so explicit directional isolates can in fact be used by higher level protocols such as CSS under implementations of the bidirectional algorithm compliant with Unicode 6.2.1.

On web pages, the explicit bidi controls (of all types - embedding, override, and isolate) should be replaced by using the dir attribute and the elements BDI and BDO. For more information, see [\[UTR20\]](#).

Although the term *embedding* is used for some explicit codes, the text within the scope of the embedding codes is not independent of the surrounding text. Characters within an embedding can affect the ordering of characters outside, and vice versa. This is not the case with the isolate codes. Characters within an isolate can not affect the ordering of characters outside it, or vice versa. That is the precise difference between the isolate and embedding codes. 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.

2.1 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 there were a Hebrew phrase in the middle of the English quotation, the that phrase could be marked as being embedded right-to-left. These codes allow for nested embeddings.

Abbr. Code	Chart Name	Description
LRE U+202A	 LEFT-TO-RIGHT EMBEDDING	Treat the following text as embedded left-to-right.
RLE U+202B	 RIGHT-TO-LEFT EMBEDDING	Treat the following text as embedded right-to-left.

The effect of right-left line direction, for example, can be accomplished by embedding the text with RLE...PDF.

2.2 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. These characters are to be avoided wherever possible, because of security concerns. For more information, see [\[UTR36\]](#).

Abbr. Code	Chart Name	Description
LRO U+202D	 LEFT-TO-RIGHT OVERRIDE	Force following characters to be treated as strong left-to-right characters.
RLO U+202E	 RIGHT-TO-LEFT OVERRIDE	Force following characters to be treated as strong right-to-left characters.

The precise meaning 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.

2.3 Terminating Explicit Directional Embeddings and Overrides

The following code terminates the effects of the last explicit embedding or override code whose effects have not yet been terminated and restores the bidirectional state to what it was before that code was encountered.

Abbr. Code	Chart Name	Description
PDF U+202C	 POP DIRECTIONAL FORMATTING	Restore the bidirectional state to what it was before the last LRE, RLE, RLO, or LRO.

The precise meaning of this code will be made clear in the discussion of the algorithm.

2.4 Explicit Directional Isolates

The following codes signal that a piece of text is to be treated as directionally isolated from its surroundings. They are very similar to the embedding codes, and like them allow for nesting. The difference between isolates and embeddings is that while an embedding roughly has the effect of a strong character on the ordering of the surrounding text, an isolate has the effect of a neutral like U+FFFC OBJECT REPLACEMENT CHARACTER. Furthermore, the text inside the isolate has no effect on the ordering of the text outside it, and vice-versa.

In addition to allowing embedding text whose direction is the opposite of its surroundings without unduly affecting its surroundings, one of the isolate codes also offers an extra feature: embedding text while inferring its direction heuristically from its constituent characters.

Abbr. Code	Chart	Name	Description
LRI Proposed U+2066	[IMAGE]	LEFT-TO-RIGHT ISOLATE	Treat the following text as isolated and left-to-right.
RLI Proposed U+2067	[IMAGE]	RIGHT-TO-LEFT ISOLATE	Treat the following text as isolated and right-to-left.
FSI Proposed U+2068	[IMAGE]	FIRST-STRONG ISOLATE	Treat the following text as isolated and in the direction of its first strong directional character that is not inside a nested isolate.

The precise meaning of these codes will be made clear in the discussion of the algorithm.

2.5 Terminating Explicit Directional Isolates

The following code terminates the effects of the last explicit directional isolate code whose effects have not yet been terminated and restores the bidirectional state to what it was before that code was encountered.



It also terminates the effects of any explicit directional embedding or override codes that came after the last directional isolate code and whose effects have not yet been terminated.

Abbr. Code	Chart	Name	Description
PDI Proposed U+2069	[IMAGE]	POP DIRECTIONAL ISOLATE	Restore the bidirectional state to what it was before the last LRI, RLI, or FSI.

The precise meaning of this code will be made clear in the discussion of the algorithm.

2.6 Implicit Directional Marks

These characters are very light-weight codes. They act exactly like right-to-left or left-to-right characters, except that they do not display or have any other semantic effect. Their use is more convenient than using explicit embeddings or overrides because their scope is much more local.

Abbr. Code	Chart	Name	Description
LRM U+200E		LEFT-TO-RIGHT MARK	Left-to-right zero-width character
RLM U+200F		RIGHT-TO-LEFT MARK	Right-to-left zero-width character

There is no special mention of the implicit directional marks in the following algorithm. That is 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.

3 Basic Display Algorithm

The Unicode Bidirectional Algorithm (UBA) takes a stream of text as input and proceeds in four main phases:

- **Separation into paragraphs.** The rest of the algorithm is applied separately to the text within each paragraph.
- **Initialization.** A list of `bidi`directional character types is initialized, with one entry for each character in the original text. The value of each entry is the `X_Bidi_Class` property of the respective character. After this point, the original characters are no longer referenced until the reordering phase. A list of embedding levels, with one level per character, is then initialized.
- **Resolution of the embedding levels.** A series of rules are applied to the lists of embedding levels and `bidi`directional character types. Each rule is based on the current values of those lists, and can modify those values. ~~Each rule is applied to each of the values in sequence before continuing to the next rule.~~ The result of this phase is a modified list of embedding levels; the list of `bidi`directional character types is no longer needed.
- **Reordering.** The text within each paragraph is reordered for display: first, the text in the paragraph is broken into lines, then the resolved embedding levels are used to reorder the text of each line for display.

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 the Paragraph Separator or appropriate Newline Function (for guidelines on the handling of CR, LF, and CRLF, see *Section 4.4, Directionality*, and *Section 5.8, Newline Guidelines* of [Unicode](#)). 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 always attach 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 attach 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, attach to the glyph on the left, or on the right, or above.

This annex uses the numbering conventions for normative definitions and rules in *Table 1*.

Table 1. 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

3.1 Definitions

BD1. The *bidirectional characters types* are values assigned to each Unicode character, including unassigned characters. The formal property name in the *Unicode Character Database [UCD]* is `X_Bidi_Class`.

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 `embedding format codes, isolate format codes and override 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 muddled with more than a small number of embeddings.

BD3. The default direction of the current embedding level (for the 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 the following 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 to be reset. The override status is set by using explicit directional controls. This status has three states, as shown in *Table 2*.

Table 2. 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

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 (a level run is also known as a *directional run*).

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), and 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: 0000000111111110000000000

Notice that the neutral character (space) between THE and CAR gets the level of the surrounding characters. The level of the neutral characters can also be changed by inserting appropriate directional marks around neutral characters. These marks have no other effects.

BD8. The *matching PDI* for a given FSI, LRI, or RLI is the one determined by the following algorithm:

- Initialize a counter to one.
- Scan the text following the FSI, LRI, or RLI to the end of the paragraph while incrementing the counter at every FSI, LRI, or RLI, and decrementing it at every PDI.
- Stop at the first PDI, if any, for which the counter is decremented to zero.
- If such a PDI was found, it is the matching PDI for the FSI, LRI, or RLI. Otherwise, there is no matching PDI for it.

Note that LRE, RLE, LRO, RLO and PDF characters are ignored when finding the matching PDI.

As we will see, an FSI, LRI, or RLI and its matching PDI are always assigned the same explicit embedding level.

BD9. An *isolating run sequence* is an ordered set of level runs where all of the following conditions are true:

- For every level run except the last one in the sequence, the last character in the level run is an FSI, LRI or RLI.
- For every level run except the first one in the sequence, the first character in the level run is the matching PDI of the FSI, LRI, or RLI at the end of the preceding level run in the sequence.
- If the first character of the first level run in the sequence is a PDI, it is not the matching PDI for any FSI, LRI, or RLI.
- If the last character of the last level run in the sequence is an FSI, LRI or RLI, it has no matching PDI.

Since the last character in each level run in an isolating run sequence (except for the last run of the sequence) is an FSI, LRI, or RLI, and the first character of the following level run is the matching PDI, which will have the same embedding level as the FSI, LRI, or RLI it matches, all the characters in an isolating run sequence will have the same explicit embedding level.

Example

Original text, assuming that no $text_i$ contains format codes or paragraph separators:

$text_1 \cdot \text{FSI} \cdot text_2 \cdot \text{LRI} \cdot text_3 \cdot \text{PDI} \cdot text_4 \cdot \text{PDI} \cdot \text{RLI} \cdot text_5 \cdot \text{PDI} \cdot text_6$

Isolating run sequences, where each cell is a level run:

- | | | |
|---------------------------|-------------------------------|---------------------------|
| $text_1 \cdot \text{FSI}$ | $\text{PDI} \cdot \text{RLI}$ | $\text{PDI} \cdot text_6$ |
|---------------------------|-------------------------------|---------------------------|
- | | |
|---------------------------|---------------------------|
| $text_2 \cdot \text{LRI}$ | $\text{PDI} \cdot text_4$ |
|---------------------------|---------------------------|
- | |
|----------|
| $text_3$ |
|----------|
- | |
|----------|
| $text_5$ |
|----------|

BD10. The *directional isolate status* is a Boolean value reflecting whether the current embedding level was started by an FSI, LRI, or RLI (as opposed to an LRE, RLE, LRO, or RLO).

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

Table 3. Abbreviations for Examples and Internal Types

Symbol	Description
N	Neutral or Separator or Isolate formatting code (B, S, WS, ON, FSI, LRI, RLI, PDI)

e	The text ordering type (L or R) that matches the embedding level direction (even or odd)
sos	The text ordering type (L or R) assigned to the virtual position before an isolating run sequence.
eos	The text ordering type (L or R) assigned to the virtual position after an isolating run sequence.

3.2 Bidirectional Character Types

The normative bidirectional character types for each character are specified in the [Unicode Character Database \[UCD\]](#) and are summarized in [Table 4](#). This is a summary only: there are exceptions to the general scope. For example, certain characters such as U+0CBF KANNADA VOWEL SIGN are given Type L (instead of NSM) to preserve canonical equivalence.

- The term European digits is used to refer to decimal forms common in Europe and elsewhere, and Arabic-Indic digits to refer to the native Arabic forms. (See [Section 8.2, Arabic](#) of [\[Unicode\]](#), for more details on naming digits.)
- Unassigned characters are given strong types in the algorithm. This 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. For assignments to character types, see [DerivedXBidiClass.txt \[DerivedXBIDI\]](#) in the [\[UCD\]](#).
- 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 U+FFFC OBJECT REPLACEMENT CHARACTER.
- As of Unicode 6.2.1, the Bidirectional Character Types are derived from the X_Bidi_Class character property instead of the Bidi_Class property used in earlier versions. The difference between the two is that X_Bidi_Class includes the LRI, RLI, FSI, and PDI bidirectional character types, which are not defined in Bidi_Class. In Unicode 7.0, which will introduce the LRI, RLI, FSI and PDI codepoints, the X_Bidi_Class and Bidi_Class values will be the same for all codepoints except for LRI, RLI, FSI, and PDI. For forward compatibility, the Bidi_Class of the RLI codepoint will be RLE, of LRI and FSI will be LRE, and of PDI will be PDF. Their X_Bidi_Class values will of course be RLI, LRI, FSI, and PDI respectively.
- As of Unicode 4.0, the Bidirectional Character Types of a few Indic characters were altered so that the Bidirectional Algorithm preserves canonical equivalence. That is, two canonically equivalent strings will result in equivalent ordering after applying the algorithm. This invariant will be maintained in the future.

Note: The Bidirectional Algorithm does *not* preserve compatibility equivalence.

Table 4. Bidirectional Character Types

Category	Type	Description	General Scope
Strong	L	Left-to-Right	LRM, most alphabetic, syllabic, Han ideographs, non-European or non-Arabic digits, ...
	R	Right-to-Left	RLM, Hebrew alphabet, and related punctuation
	AL	Right-to-Left Arabic	Arabic, Thaana, and Syriac alphabets, most punctuation specific to those scripts, ...
Weak	EN	European Number	European digits, Eastern Arabic-Indic digits, ...
	ES	European Number Separator	PLUS SIGN, MINUS SIGN
	ET	European Number Terminator	DEGREE SIGN, 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	Default ignorables, non-characters, and control characters, other than those explicitly given other types.
Neutral	B	Paragraph Separator	PARAGRAPH SEPARATOR, appropriate Newline Functions, higher-level 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
Explicit Formatting	LRE	Left-to-Right Embedding	LRE
	LRO	Left-to-Right Override	LRO
	RLE	Right-to-Left Embedding	RLE
	RLO	Right-to-Left Override	RLO
	PDF	Pop Directional Format	PDF
	LRI	Left-to-Right Isolate	LRI
	RLI	Right-to-Left Isolate	RLI
	FSI	First-Strong Isolate	FSI
	PDI	Pop Directional Isolate	PDI

3.3 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.

3.3.1 The Paragraph Level

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

P2. In each paragraph, find the first character of type L, AL, or R while skipping over any characters between an FSI, LRI or RLI and its matching PDI or, if it has no matching PDI, the end of the paragraph.

Because paragraph separators delimit text in this algorithm, this will generally be the first strong character after a paragraph separator or at the very beginning of the text. Note that the characters of type LRE, LRO, RLE, or RLO are ignored in this rule. This is because typically they are used to indicate that the embedded text is the *opposite* direction than the paragraph level.

P3. If a character is found in [P2](#) and it is of type AL or R, then set the paragraph embedding level to one; otherwise, set it to zero.

Whenever a higher-level protocol specifies the paragraph level, rules [P2](#) and [P3](#) may be overridden: see [HL1](#).

3.3.2 Explicit Levels and Directions

All explicit embedding levels are determined from the embedding, override and isolate codes, by applying the explicit level rules [X1](#) through [X9](#). These rules are applied as part of the same logical pass over the input. The following variables are used during this pass:

- The current embedding level.
- The current directional override status.
- The current directional isolate status.
- A directional status stack of at most 61 entries where each entry consists of:
 - An embedding level.
 - A directional override status.
 - A directional isolate status.
- A counter called the *valid isolate count*.
- A counter called the *invalid isolate count*.
- A counter called the *invalid embedding count*.

As each character is processed, these variables' values are modified and the character's explicit embedding level is set as defined by rules [X1](#) through [X9](#) on the basis of the character's bidirectional type and the variables' current values.

Explicit Embeddings

X1. Begin by setting the current embedding level to the paragraph embedding level. Set the current directional override status to neutral. Set the current directional isolate status to false. Set the stack to empty. Set the valid isolate count to zero. Set the invalid isolate count to zero. Set the invalid embedding count to zero. 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, and the invalid isolate count and invalid embedding count are both zero, then this RLE is valid. Push the current embedding level, override status and isolate status on the directional status stack. Reset the current embedding level to the new level, reset the current isolate status to **false**, and reset the current override status to **neutral**.

b. Otherwise, this RLE is invalid. Do not change the current level, override status or isolate status. If the invalid isolate count is zero, increment the invalid embedding count by one.

For example, assuming the invalid counts are both zero, level 0 → 1; levels 1, 2 → 3; levels 3, 4 → 5; ...59, 60 → 61. Above 60 or if either invalid count is non-zero, the RLE is invalid.

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

a. If this new level would be valid, and the invalid isolate count and invalid embedding count are both zero, then this LRE is valid. Push the current embedding level, override status and isolate status on the directional status stack. Reset the current embedding level to the new level, reset the current isolate status to **false**, and reset the current override status to **neutral**.

b. Otherwise, this LRE is invalid. Do not change the current level, override status or isolate status. If the invalid isolate count is zero, increment the invalid embedding count by one.

For example, assuming the invalid counts are both zero, levels 0, 1 → 2; levels 2, 3 → 4; levels 4, 5 → 6; ...58, 59 → 60. Above 59 or if either invalid count is non-zero, the LRE is invalid.

Explicit Overrides

An explicit directional override sets the embedding level in the same way the explicit embedding codes do, but also changes the bidirectional 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, and the invalid isolate count and invalid embedding count are both zero, then this RLO is valid. Push the current embedding level, override status and isolate status on the directional status stack. Reset the current embedding level to the new level, reset the current isolate status to **false**, and reset the current override status to **right-to-left**.

b. Otherwise, this RLO is invalid. Do not change the current level, override status or isolate status. If the invalid isolate count is zero, increment the invalid embedding count by one.

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

a. If this new level would be valid, and the invalid isolate count and invalid embedding count are both zero, then this LRO is valid. Push the current embedding level, override status and isolate status on the directional status stack. Reset the current embedding level to the new level, reset the current isolate status to **false**, and reset the current override status to **left-to-right**.

b. Otherwise, this LRO is invalid. Do not change the current level, override status or isolate status. If the invalid isolate count is zero, increment the invalid embedding count by one.

Isolates

X5a. With each RLI, first set the RLI's level to the current embedding level, and then compute the least greater **odd** embedding level.

a. If this new level would be valid and the invalid isolate count and the invalid embedding count are both zero, then this RLI is valid. Increment the valid isolate count by one. Push the current embedding level, override status, and isolate status on the directional status stack. Reset the current embedding level to the new level, reset the current isolate status to **true**, and reset the current override status to **neutral**.

b. Otherwise, this RLI is invalid. Increment the invalid isolate count by one, and leave all other variables unchanged.

X5b. With each LRI, first set the LRI's level to the current embedding level, and then compute the least greater **even** embedding level.

a. If this new level would be valid and the invalid isolate count and the invalid embedding count are both zero, then this LRI is valid. Increment the valid isolate count by one. Push the current embedding level, override status, and isolate status on the directional status stack. Reset the current embedding level to the new level, reset the current isolate status to **true**, and reset the current override status to **neutral**.

b. Otherwise, this LRI is invalid. Increment the invalid isolate count by one, and leave all other variables unchanged.

X5c. With each FSI, apply rules [P2](#) and [P3](#) to the sequence of characters between the FSI and its matching PDI, or if there is no matching PDI, the end of the paragraph, as if this sequence of characters were a paragraph. If these rules decide on paragraph embedding level 1, treat the FSI as an RLI in rule [X5a](#). Otherwise, treat it as an LRI in rule [X5b](#).

Note that the current embedding level is not reset to the paragraph embedding level determined by P2 and P3. It goes up by one or two levels, as it would for an LRI or RLI.

Non-formatting characters

X6. For all types besides BN, RLE, LRE, RLO, LRO, PDF, RLI, LRI, FSI, and PDI:

- a. Set the level of the current character to the current embedding level.
- b. Whenever the current directional override status is not neutral, reset the current character type according to the current directional override status.

If the current 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. The current embedding level is not changed by this rule.

Terminating Isolates

There is a single bidirectional character type, PDI, to terminate the scope of the last unterminated FSI, LRI, or RLI. It also terminates the scopes of all unterminated LREs, RLEs, LROs, and RLOs appearing after that last unterminated FSI, LRI, or RLI.

X6a. With each PDI, perform the following steps:

- a. If the invalid isolate count is greater than zero, decrement it by one. (This PDI matches an invalid FSI, LRI, or RLI.)

- b. Otherwise, if the valid isolate count is zero, do nothing. (This PDI does not match any FSI, LRI, or RLI, valid or invalid.)

- c. Otherwise, perform the following steps. (This PDI matches a valid FSI, LRI, or RLI.)

- c.1. Reset the invalid embedding count to zero. (This terminates the scope of those invalid LREs, RLEs, LROs and RLOs encountered after the FSI, LRI, or RLI matched by the PDI whose scopes have not been terminated by a matching PDF.)

- c.2. While the current directional isolate status is false, keep popping the last entry from the directional status stack into the current embedding level, current directional override status and current directional isolate status. (This terminates the scope of those valid LREs, RLEs, LROs, and RLOs encountered after the FSI, LRI, or RLI matched by the PDI whose scopes have not been terminated by a matching PDF. It leaves the current directional isolate status true, in the scope of the matched FSI, LRI, or RLI.)

- c.3. Pop the last entry from the directional status stack into the current embedding level, current directional override status and current directional isolate status, and decrement the valid isolate count by one. (This terminates the scope of the matched FSI, LRI, or RLI. Given that the valid isolate count is non-zero, the stack could not have been empty.)

- d. In all cases, set the PDI's level to the current embedding level (i.e. the one after

the steps above).

Note that the level assigned to an FSI, LRI, or RLI is always the same as that assigned to the matching PDI.

Terminating Embeddings and Overrides

There is a single `bidirectional character type, PDF`, to terminate the scope of the `current explicit code, whether an embedding or a directional override` last unterminated LRE, RLE, LRO, or RLO. However, the PDF only terminates it if the last unterminated FSI, LRI, or RLI came before the last unterminated LRE, RLE, LRO, or RLO. Otherwise, the PDF is ignored.

X7. With each PDF, determine the matching embedding or override code. If there was a valid matching code, restore (pop) the last remembered (pushed) embedding level and directional override; perform the following steps:

a. If the invalid isolate count is greater than zero, do nothing. (This PDF is within the scope of an invalid FSI, LRI or RLI. It either matches and terminates the scope of a subsequent and thus invalid LRE, RLE, LRO, or RLO, or does not match any LRE, RLE, LRO, or RLO.)

b. Otherwise, if the invalid embedding count is greater than zero, decrement it by one. (This PDF matches and terminates the scope of an invalid LRE, RLE, LRO, or RLO.)

c. Otherwise, if the current directional isolate status is false, pop the last entry from the directional status stack into the current embedding level, current directional override status and current directional isolate status. (This PDF matches and terminates the scope of a valid LRE, RLE, LRO, or RLO.)

d. Otherwise, do nothing. (This PDF does not match any LRE, RLE, LRO, or RLO.)

End of Paragraph

All codes and pushed states are completely popped at the end of a paragraph.

*X8. All explicit directional embeddings, overrides and isolates are completely terminated at the end of each paragraph. Paragraph separators are **not** included in any embedding, override or isolate, and are thus assigned the paragraph embedding level.*

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 Section 5, [Implementation Notes](#), for information on implementing the algorithm without removing the formatting codes.

- The *zero width joiner* and *non-joiner* affect the shaping of the adjacent characters —those that are adjacent in the original backing-store order, even though those characters may end up being rearranged to be non-adjacent by the Bidirectional Algorithm. For more information, see Section 5.3, [Joiners](#).
- Note that FSI, LRI, RLI, and PDI codes are **not** removed. As indicated by the rules below, they are used to determine the paragraphs's isolating run sequences, within which they are then treated as neutral characters (as implied by their inclusion in the N symbol defined in Table 3).

X10. The remaining rules are applied to each isolating run sequence. The rules must be applied to a given isolating run sequence in the order in which the rules appear below, applying one rule to all the characters in the sequence in the order in which they appear in the sequence, before starting to apply another rule. When applying a rule to an isolating run sequence, the last value of each level run in the isolating run sequence is treated as if it were immediately followed by the first value in the next level run in the sequence, if any. In order to apply the rules, determine the start-of-sequence (sos) and end-of-sequence (eos) types, either L or R, for each isolating run sequence. These depend on the higher of the two levels on either side of the sequence boundary. That is, for sos, compare the level of the first character in the sequence with the level of the character preceding it in the paragraph, and if there is none, with the paragraph embedding level. For eos, compare the level of the last character in the sequence with the level of the character following it in the paragraph, and if there is none, with the paragraph embedding level. If the higher level is odd, the sos or eos 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, sos is L, eor is R.
 Run 2 is at level 1, sos is R, eor is L.
 Run 3 is at level 2, sos is L, eor is L.

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

Let's take a look at some examples, each of which is assumed to be a paragraph with base level 0 where no text_i contains formatting codes or paragraph separators.

Example 1: text₁ · RLE · text₂ · LRE · text₃ · PDF · text₄ · PDF · RLE · text₅ · PDF · text₆

Isolating Run Sequence	Embedding Level	sos	eos
text ₁	0	L	R

text ₂	1	R	L
text ₃	2	L	L
text ₄ · text ₅	1	L	R
text ₆	0	R	L

Example 2: text₁ · **RLI** · text₂ · **LRI** · text₃ · **PDI** · text₄ · **PDI** · **RLI** · text₅ · **PDI** · text₆

Isolating Run Sequence	Embedding Level	sos	eos
text ₁ · RLI · PDI · RLI · PDI · text ₆	0	L	L
text ₂ · LRI · PDI · text ₄	1	R	R
text ₃	2	L	L
text ₅	1	R	R

Example 3: text₁ · **RLE** · text₂ · **LRI** · text₃ · **RLE** · text₄ · **PDI** · text₅ · **PDF** · text₆

Isolating Run Sequence	Embedding Level	sos	eos
text ₁	0	L	R
text ₂ · LRI · PDI · text ₅	1	R	R
text ₃	2	L	R
text ₄	3	R	R
text ₆	0	R	L

3.3.3 Resolving Weak Types

Weak types are now resolved one **isolating run sequence** at a time. At **isolating run sequence** boundaries where the type of the character on the other side of the boundary is required, the type assigned to **sos** or **eos** is used.

Nonspacing marks are now resolved based on the previous characters.

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

Assume in this example that **sos** is R:

AL NSM NSM → AL AL AL

$s\boxed{s}$ NSM → $s\boxed{s}$ 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 $s\boxed{s}$) 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

$s\boxed{s}$ N EN → $s\boxed{s}$ 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 **so**) 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

3.3.4 Resolving Neutral Types

Neutral types are now resolved one isolating run sequence at a time. At isolating run sequence boundaries where the type of the character on the other side of the boundary is required, the type assigned to **so** or **eo** 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. Paired punctuation marks are considered as a pair so that they both resolve to the same direction. European and Arabic numbers act as if they were R in terms of their influence on neutrals.

N0. Paired punctuation marks take the embedding direction if the enclosed text contains mixed strong types or a strong type of the embedding direction only. Else, if the enclosed text contains a strong type of the opposite direction only, and at least one external neighbor also has that direction, the paired punctuation marks take the direction opposite the embedding direction.

Paired punctuation marks are pairs of characters A and B, where A has general category Open_Punctuation (gc = Ps), B has general category Close_Punctuation (gc = Pe), and A and B form a mirrored pair (Bidi_Mirrored = Yes for both, and Bidi_Mirroring_Glyph of A is B).

This rule is applied to paired punctuation marks that are correctly nested. When paired punctuation marks are mismatched, pairing occurs between the closest pairable marks in logical order.

Example 1 - Enclosed mixed strong types (RTL paragraph direction)

Storage:	smith		(fabrikam		ARABIC)		HEBREW
Bidi class:	L	WS	ON	L	WS	R	ON	WS	R
Rules applied:		N2->R	N0->R		N2->R		N0->R	N1->R	
Resolved level:	2	2	1	2	1	1	1	1	1
Display:	WERBEH (CIBARA fabrikam) smith								

Example 2 - Enclosed strong type opposite the embedding direction (RTL paragraph direction)

Storage:	ARABIC		book	(s)
----------	--------	--	------	---	---	---

Bidi class:	R	WS	L	ON	L	ON
Rules applied:		N2->R		N0->L		N0->L
Resolved level:	1	1	2	2	2	2
Display:	book (s) CIBARA					

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 act as if they were R in terms of their influence on neutrals. Start-of-level-run (sos) and end-of-level-run (eos) are used at isolating run sequence boundaries.

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

N2. Any remaining neutrals take the embedding direction.

N → e

The embedding direction for the given neutral character is derived from its embedding level: L if the character is set to an even level, and R if the level is odd. (See [BD3](#).)

Assume in the following example that eos is L and sos is R. Then an application of [N1](#) and [N2](#) yields the following:

L N eos → L L eos
R N eos → R e eos
sos N L → sos e L
sos N R → sos 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 by digits (see *Section 3.3.3, Resolving Weak Types*). However, if there is a preceding left-to-right sequence, then European numbers will adopt that direction:

Storage: IT IS A bmw 500, OK.

Display: .KO ,bmw 500 A SI TI

3.3.5 Resolving Implicit Levels

In the final phase, the embedding level of text may be increased, based on 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 results in the following rules:

11. 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.

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

Table 5 summarizes the results of the implicit algorithm.

Table 5. 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

3.4 Reordering Resolved Levels

The following rules describe the logical process of finding the correct display order. As opposed to resolution phases, these rules act on a per-line basis *and are applied after any line wrapping is applied to the paragraph.*

Logically there are the following steps:

- The levels of the text are determined according to the previous rules.
- 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 and/or isolate format codes (FSI, LRI, RLI, and PDI) preceding a segment separator or paragraph separator, and*
4. *Any sequence of whitespace characters and/or isolate format codes (FSI, LRI, RLI, and PDI) 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 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, including intermediate levels not actually present in the text, reverse any contiguous sequence of characters that are at that level or higher.

This rule reverses a progressively larger series of substrings.

The following examples illustrate the reordering, showing the successive steps in application of Rule [L2](#). The original text, including any embedding codes for producing the particular levels, is shown in the "Storage" row in the example tables. The application of the rules from [Section 3.3 Resolving Embedding Levels](#) and of the Rule [L1](#) results in (a) text with BN characters and some Bidi Controls removed, plus (b) resolved levels. These are listed in the rows "Before Reordering" and "Resolved Levels". Each successive row thereafter shows the one pass of reversal from Rule [L2](#), such as "Reverse levels 1-2". At each iteration, the underlining shows the text that has been reversed.

The paragraph embedding level for the first and third examples is 0 (left-to-right direction), and for the second and fourth examples is 1 (right-to-left direction).

Example 1 (embedding level = 0)

Storage: `car means CAR.`

Before Reordering: `car means CAR.`

Resolved levels: 00000000001110

Reverse level 1: car means RAC.

Example 2 (embedding level = 1)

Storage: car MEANS CAR.



Before Reordering: car MEANS CAR.

Resolved levels: 22211111111111

Reverse level 2: rac MEANS CAR.

Reverse levels 1-2: .RAC SNAEM car

Example 3 (embedding level = 0)

Storage: he said "car MEANS CAR.

Before Reordering: he said "car MEANS CAR."

Resolved levels: 0000000022211111111100

Reverse level 2: he said "rac MEANS CAR."

Reverse levels 1-2: he said "RAC SNAEM car."

Example 4 (embedding level = 1)

Storage: DID YOU SAY 'he said "car MEANS CAR

Before Reordering: DID YOU SAY 'he said "car MEANS CAR"'

Resolved levels:: 1111111111112222222244433333333211

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 (thus 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 (thus 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 5.13, Rendering Nonspacing Marks* of [\[Unicode\]](#), for more information.

L4. A character is depicted by a mirrored glyph if and only if (a) the resolved directionality of that character is R, and (b) the Bidi_Mirrored property value of that character is true.

- *The Bidi_Mirrored property is defined by Section 4.7, Bidi Mirrored—Normative of [\[Unicode\]](#); the property values are specified in [\[UCD\]](#).*
- *This rule can be overridden in certain cases; see [HL6](#).*

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. Note that for backward compatibility the characters U+FD3E (()) ORNATE LEFT PARENTHESIS and U+FD3F ()) ORNATE RIGHT PARENTHESIS are not mirrored.

3.5 Shaping

Cursively connected scripts, such as Arabic or Syriac, require the selection of positional character shapes that depend on adjacent characters (see *Section 8.2, Arabic* of [\[Unicode\]](#)). Shaping is logically applied *after* the Bidirectional Algorithm is used and is limited to characters within the same directional run. Consider the following example string of Arabic characters, which is represented in memory as characters 1, 2, 3, and 4, and where the first two characters are overridden to be LTR. To show both paragraph directions, the next two are embedded, but with the normal RTL direction.

1	2	3	4
ج	ع	ل	م
062C JEEM	0639 AIN	0644 LAM	0645 MEEM
L	L	R	R

One can use embedding codes to achieve this effect in plain text or use markup in HTML, as in the examples below. (The **bold** text would be for the right-to-left paragraph direction.)

- LRM/RLM LRO *JEEM AIN* PDF RLO *LAM MEEM* PDF
- `<p dir="ltr"/>LRO JEEM AIN PDF RLO LAM MEEM PDF</p>`
- `<p dir="ltr"/><bdo dir="ltr">JEEM AIN</bdo>
<bdo dir="rtl">LAM MEEM</bdo></p>`

The resulting shapes will be the following, according to the paragraph direction:

Left-Right Paragraph				Right-Left Paragraph			
1	2	4	3	4	3	1	2
ج	ع	م	ل	م	ل	ج	ع
JEEM-F	AIN-I	MEEM-F	LAM-I	MEEM-F	LAM-I	JEEM-F	AIN-I

3.5.1 Shaping and Line Breaking

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, the width calculations must be based on the shaped glyphs.

Note that the *soft-hyphen* (SHY) works in cursively connected scripts as it does in other scripts. That is, it indicates a point where the line could be broken in the middle of a word. If the rendering system breaks at that point, the display—including shaping—should be what is appropriate for the given language. For more information on this and other line breaking issues, see Unicode Standard Annex #14, “Line Breaking Properties” [UAX14].

4 Bidirectional Conformance

A process that claims conformance to this specification shall satisfy the following clauses:

UAX9-C1.

In the absence of a permissible higher-level protocol, a process that renders text shall display all visible representations of characters (excluding format characters) in the order described by Section 3, [Basic Display Algorithm](#), of this annex. In particular, this includes definitions [BD1-BD7](#) and steps [P1-P3](#), [X1-X10](#), [W1-W7](#), [N1-N2](#), [I1-I2](#), and [L1-L4](#).

- As is the case for all other Unicode algorithms, this is a *logical* description —particular implementations can have more efficient mechanisms as long as they produce the same results. See C18 in *Chapter 3, Conformance* of [Unicode], and the notes following.
- The Bidirectional Algorithm specifies part of the intrinsic semantics of right-to-left characters and is thus required for conformance to the Unicode Standard where any such characters are displayed.

UAX9–C2.

The only permissible higher-level protocols are those listed in Section 4.3, [Higher-Level Protocols](#). They are [HL1](#), [HL2](#), [HL3](#), [HL4](#), [HL5](#), and [HL6](#).

Use of higher-level protocols is discouraged, because it introduces interchange problems and can lead to security problems. For more information, see Unicode Technical Report #36, “Unicode Security Considerations” [UTR36].

4.1 Boundary Neutrals

The goal in marking a format or control character as BN is that it have no effect on the rest of the algorithm. (ZWJ and ZWNJ are exceptions; see [X9](#)). Because conformance does not require the precise ordering of format characters with respect to others, implementations can handle them in different ways as long as they preserve the ordering of the other characters.

4.2 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 [four](#) classes:

- *No bidirectional formatting*. This 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.
- *Non-isolate bidirectionality*. The implicit Bidirectional Algorithm, the implicit directional marks, and the explicit non-isolate directional embedding codes are supported: RLM, LRM, LRE, RLE, LRO, RLO, PDF.
- *Full bidirectionality*. The implicit Bidirectional Algorithm, the implicit directional marks, and [all](#) the explicit directional embedding codes are supported: RLM, LRM, LRE, RLE, LRO, RLO, PDF, [FSI](#), [LRI](#), [RLI](#), [PDI](#).

4.3 Higher-Level Protocols

The following clauses are the only permissible ways for systems to apply higher-level protocols to the ordering of bidirectional text. Some of the clauses apply to *segments* of

structured text. This refers to the situation where text is interpreted as being structured, whether with explicit markup such as XML or HTML, or internally structured such as in a word processor or spreadsheet. In such a case, a segment is span of text that is distinguished in some way by the structure.

HL1. Override [P3](#), and set the paragraph embedding level explicitly.

- A higher-level protocol may set any paragraph level. This can be done on the basis of the context, such as on a table cell, paragraph, document, or system level. ([P2](#) may be skipped if [P3](#) is overridden). Note that this does not allow a higher-level protocol to override the limit specified in [BD2](#).
- A higher-level protocol may apply rules equivalent to [P2](#) and [P3](#) but default to level 1 (RTL) rather than 0 (LTR) to match overall RTL context.
- A higher-level protocol may use an entirely different algorithm that heuristically auto-detects the paragraph embedding level based on the paragraph text and its context. For example, it could base it on whether there are more RTL characters in the text than LTR. As another example, when the paragraph contains no strong characters, its direction could be determined by the levels of the paragraphs before and after.

HL2. Override [W2](#), and set EN or AN explicitly.

- A higher-level protocol may reset characters of type EN to AN, or vice versa, and ignore [W2](#). For example, style sheet or markup information can be used within a span of text to override the setting of EN text to be always be AN, or vice versa.

HL3. Emulate [explicit](#) directional [formatting](#) codes.

- A higher-level protocol can impose a directional embedding, [isolate](#) or override on a segment of structured text. The behavior must always be defined by reference to what would happen if the equivalent explicit codes as defined in the algorithm were inserted into the text. For example, a style sheet or markup can set the embedding level on a span of text.

HL4. Apply the Bidirectional Algorithm to segments.

- The Bidirectional Algorithm can be applied independently to one or more segments of structured text. For example, when displaying a document consisting of textual data and visible markup in an editor, a higher-level process can handle syntactic elements in the markup separately from the textual data.

HL5. Provide artificial context.

- Text can be processed by the Bidirectional Algorithm as if it were preceded by a character of a given type and/or followed by a character of a given type. This allows a piece of text that is extracted from a longer sequence of text to behave as it did in the larger context.

HL6. Additional mirroring.

- Certain characters that do not have the Bidi_Mirrored property can also be depicted by a mirrored glyph in specialized contexts. Such contexts include, but are not limited to, historic scripts and associated punctuation, private-use characters, and characters in mathematical expressions. (See Section 6, [Mirroring](#).) These characters are those that fit at least one of the following conditions:
 1. Characters with a resolved directionality of R
 2. Characters with a resolved directionality of L and whose bidi class is R or AL

Clauses [HL1](#) and [HL3](#) are specialized applications of the more general clauses [HL4](#) and [HL5](#). They are provided here explicitly because they directly correspond to common operations.

As an example of the application of [HL4](#), suppose an XML document contains the following fragment. (Note: This is a simplified example for illustration: element names, attribute names, and attribute values could all be involved.)

```
ARABICenglishARABIC<e1 type='ab'>ARABICenglish<e2 type='cd'>english
```

This can be analyzed as being five different segments:

a. ARABICenglishARABIC

- b. `<e1 type='ab'>`
- c. ARABICenglish
- d. `<e2 type='cd'>`
- e. english

To make the XML file readable as source text, the display in an editor could order these elements all in a uniform direction (for example, all left-to-right) and apply the Bidirectional Algorithm to each field separately. It could also choose to order the element names, attribute names, and attribute values uniformly in the same direction (for example, all left-to-right). For final display, the markup could be ignored, allowing all of the text (segments a, c, and e) to be reordered together.

4.4 Bidi Conformance Testing

The `BidiTest.txt` file in the *Unicode Character Database* [UCD] provides a conformance test for UBA implementations. It is designed to be reasonably compact, and yet provide a thorough test of all cases up to a given limit (currently 4). The format is described in detail in the header of the file.

5 Implementation Notes

5.1 Reference Code

There are two versions of BIDI reference code available. Both have been tested to produce identical results. One version is written in Java, and the other is written in C++. The Java version is designed to closely follow the steps of the algorithm as described below. The C++ code is designed to show one of the optimization methods that can be applied to the algorithm, using a state table for one phase.

One of the most effective optimizations is to first test for right-to-left characters and not invoke the Bidirectional Algorithm unless they are present.

There are two directories containing source code for reference implementations at [Code9]. Implementers are encouraged to use this resource to test their implementations. There is an online demo of bidi code at <http://unicode.org/cldr/utility/bidi.jsp>, which shows the results, plus the levels and the rules invoked for each character.

5.2 Retaining Format Codes

Some implementations may wish to retain the format codes when running the algorithm. The following provides a summary of how this may be done. Note that this summary is an informative implementation guideline; it should provide the same results as the explicit algorithm above, but in case of any deviation the explicit algorithm is the normative statement for conformance.

- In rule [X9](#), instead of removing the format codes, assign the embedding level to each embedding character, and turn it into BN.
- In rule [X10](#), assign L or R to the last of a sequence of adjacent BNs according to the `eos` / `sos`, and set the level to the higher of the two levels.

- In rule [W1](#), search backward from each NSM to the first character in the [isolating run sequence](#) whose type is not BN, and set the NSM to its type. If the NSM is the first non-BN character, it will get the type of [sos](#).
- 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 adjacent to ET, ES, or CS to ON as well.
- In rule [W7](#), scan past BN.
- In rules [N1](#) and [N2](#), treat BNs adjoining neutrals same as those neutrals.
- In rules [I1](#) and [I2](#), ignore BN.
- In rule [L1](#), include format codes and BN together with whitespace characters in the sequences whose level gets reset before a separator or line break. Resolve any LRE, RLE, LRO, RLO, PDF, or BN to the level of the preceding character if there is one, and otherwise to the base level.

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

5.3 Joiners

As described under [X9](#), the *zero width joiner* and *non-joiner* affect the shaping of the adjacent characters—those that are adjacent in the original backing-store order—even though those characters may end up being rearranged to be non-adjacent by the Bidirectional Algorithm. To determine the joining behavior of a particular character after applying the Bidirectional Algorithm, there are two main strategies:

- When shaping, an implementation can refer back to the original backing store to see if there were adjacent ZWNJ or ZWJ characters.
- Alternatively, the implementation can replace ZWJ and ZWNJ by an out-of-band character property associated with those adjacent characters, so that the information does not interfere with the Bidirectional Algorithm and the information is preserved across rearrangement of those characters. Once the Bidirectional Algorithm has been applied, that out-of-band information can then be used for proper shaping.

5.4 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, because 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° counterclockwise so that the characters are ordered from top to bottom. Latin text and numbers may be rotated 90° clockwise so that the characters are also ordered from top to bottom.

The Bidirectional Algorithm is used when some characters are ordered from bottom to top. For example, this happens with a mixture of Arabic and Latin glyphs when all the

glyphs are rotated uniformly 90° clockwise. The Unicode Standard does not specify whether text is presented horizontally or vertically, or whether text is rotated. That is left up to higher-level protocols.

5.5 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 there are nested segments of different-direction text, or there are weak characters on directional boundaries. In these cases, embeddings or directional marks may be required to get the right display. Part numbers may also require directional overrides.

The most common problematic case is that of neutrals on the boundary of an embedded language. This can be addressed by setting the level of the embedded text correctly. For example, with all the 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.

However, a simpler and better method of doing this is to place a right directional mark (RLM) after the exclamation mark. Because the exclamation mark is now not 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 make use of 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

5.6 Separating Punctuation Marks

A common problem case is where the text really represents a sequence of items with separating punctuation marks, often programmatically concatenated. These separators are often strings of neutral characters. For example, a web page might have the following at the bottom:

advertising programs - business solutions - privacy policy - help - about

This might be built up on the server by concatenating a variable number of strings with " - " as a separator, for example. If all of the text is translated into Arabic or Hebrew and the overall page direction is set to be RTL, then the right result occurs, such as the following:

TUOBA - PLEH - YCILOP YCAVIRP - SNOITULOS SSENISUB - SMARGORP
GNISITREVDA

However, suppose that in the translation, there remain some LTR characters. This is not uncommon for company names, product names, technical terms, and so on. If one of the separators is bounded on both sides by LTR characters, then the result will be badly jumbled. For example, suppose that "programs" in the first term and "business" in the second were left in English. Then the result would be

TUOBA - PLEH - YCILOP YCAVIRP - SNOITULOS programs - business
GNISITREVDA

The result is a jumble, with the apparent first term being "advertising business" and the second being "programs solutions". The simplest solution for this problem is to include an RLM character in each separator string. That will cause each separator to adopt a right-to-left direction, and produce the correct output:

TUOBA - PLEH - YCILOP YCAVIRP - SNOITULOS business - programs
GNISITREVDA

The stateful controls (LRE, RLE, and PDF) can be used to achieve the same effect; web pages would use spans with the attributes *dir="ltr"* or *dir="rtl"*. Each separate field would be embedded, excluding the separators. In general, LRM and RLM are preferred to the stateful approach because their effects are more local in scope, and are more robust than the *dir* attributes when text is copied. (Ideally programs would convert *dir* attributes to the corresponding stateful controls when converting to plain text, but that is not generally supported.)

5.7 Migrating from 2.0 to 3.0

In the Unicode Character Database for [Unicode3.0], new bidirectional character types were introduced to make the body of the Bidirectional Algorithm depend only on the types of characters, and not on the character values. The changes from the 2.0 bidirectional types are listed in *Table 6*.

Table 6. New Bidirectional Types in Unicode 3.0

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 adjust to the modifications in the Bidirectional Algorithm by algorithmically remapping the characters in *Table 6* to the new types.

5.8 Conversion to Plain Text

For consistent appearance, when bidirectional text subject to a higher-level protocol is to be converted to Unicode plain text, formatting codes should be inserted to ensure that the display order resulting from the application of the Unicode Bidirectional Algorithm matches that specified by the higher-level protocol.. The same principle should be followed whenever text using a higher-level protocol is converted to marked-up text that is unaware of the higher-level protocol. For example, if a higher-level protocol sets the paragraph direction to 1 (R) based on the number of L versus R/AL characters, when converted to plain text the paragraph would be embedded in a bracketing pair of RLE..PDF formatting codes. If the same text were converted to HTML4.0 the attribute `dir = "rtl"` would be added to the paragraph element.

6 Mirroring

The mirrored property is important to ensure that the correct character codes are used for the desired semantic. This is of particular importance where the name of a character does not indicate the intended semantic, such as with U+0028 “(” LEFT PARENTHESIS. While the name indicates that it is a left parenthesis, the character really expresses an *open parenthesis*—the *leading* character in a parenthetical phrase, not the trailing one.

Some of the characters that do not have the Bidi_Mirrored property may be rendered with mirrored glyphs, according to a higher level protocol that adds mirroring: see *Section 4.3, [Higher-Level Protocols](#)*, especially [HL6](#). Except in such cases, mirroring must be done according to rule [L4](#), to ensure that the correct character code is used to express the intended semantic of the character, and to avoid interoperability and security problems.

Implementing rule [L4](#) calls for mirrored glyphs. These glyphs may not be exact *graphical* mirror images. For example, clearly an italic parenthesis is not an exact mirror image of another—“(” is not the mirror image of “)”. Instead, mirror glyphs are those acceptable as mirrors within the normal parameters of the font in which they are represented.

In implementation, sometimes pairs of characters are acceptable mirrors for one another—for example, U+0028 “(” LEFT PARENTHESIS and U+0029 “)” RIGHT PARENTHESIS OF U+22E0 “∫” DOES NOT PRECEDE OR EQUAL and U+22E1 “∫” DOES NOT SUCCEED OR EQUAL. Other characters such as U+2231 “∫” CLOCKWISE INTEGRAL do not have corresponding characters that can be used for acceptable mirrors. The informative Bidi Mirroring data file [\[Data9\]](#), lists the paired characters with acceptable mirror glyphs. The formal property name for this data in the *Unicode Character Database* [\[UCD\]](#) is Bidi_Mirroring_Glyph. A comment in the file indicates where the pairs are “best fit”: they should be acceptable in rendering, although ideally the mirrored glyphs may have somewhat different shapes.

Acknowledgments

Mark Davis created the initial version of this annex and maintains the text.

Thanks to the following people for their contributions to the Bidirectional Algorithm or for their feedback on earlier versions of this annex: Aharon Lanin (אהרון לנין), Ahmed Talaat (أحمد طلعت), Alaa Ghoneim (علاء غنيم), Ahmed Talaat (أحمد طلعت), Andrew Glass, Asmus Freytag, Avery Bishop, Behdad Esfahbod (بهداد اسفهبود), Doug Felt, Dwayne Robinson, Eric Mader, Ernest Cline, Gidi Shalom-Bendor (גידי שלום-בן דור), Isai Scheinberg, Israel Gidali (ישראל גידלי), Joe Becker, John McConnell, Jonathan Kew, Jonathan Rosenne (יונתן רוזן), Khaled Sherif (خالد شريف), Kamal Mansour (كمال منصور), Kenneth Whistler, Khaled Sherif (خالد شريف), Laurentiu Iancu, Maha Hassan (مها حسن), Markus Scherer, Martin Dürst, Mati Allouche (מתתיהו אלוש), Michel Suignard, Mike Ksar (ميشيل قصار), Murray Sargent, Paul Nelson, Rick McGowan, Robert Steen, Roozbeh Pournader (روزبه پورنادر), Steve Atkin, and Thomas Milo (توماس ميلو).

References

For references for this annex, see Unicode Standard Annex #41, “[Common References for Unicode Standard Annexes](#).”

Modifications

The following summarizes modifications from previous revisions of this annex.

Revision 28

- **Proposed Update** for Unicode 6.2.1.
- Major extension of the algorithm to allow for implementation of isolates and the introduction of the X_Bidi_Class property.
- Adds [BD8](#), [BD9](#), and [BD10](#), Sections [2.4](#) and [2.5](#), and Rules [X5a](#), [X5b](#), [X5c](#) and [X6a](#).
- Modifies Rule [X10](#) to make the isolating run sequence the unit to which subsequent rules are applied.
- Extensively revises Section [3.3.2](#) and its other X rules to formalize the algorithm for matching a PDF with the embedding or override code whose scope it terminates.
- Invalidates an RLE or RLO nested inside the scope of an LRE or LRO nested inside embedding level 60 (where that LRE or LRO was already invalid). This was

required to allow nesting embeddings and overrides in isolates and vice-versa.

- Changes to section Section 3.3.4 [Resolving Neutral Types](#) to resolve paired punctuation marks as a unit. Adds [N0](#).

Revision 27

- **Reissued** for Unicode 6.2.0.

Revision 26 being a proposed update, only changes between versions 23 and 27 are noted here.

Revision 25

- **Reissued** for Unicode 6.1.0.

Revision 24 being a proposed update, only changes between versions 23 and 25 are noted here.

Revision 23

- **Reissued** for Unicode 6.0.0.
- Added anchors on tables.
- Added text to clarify [HL1](#), and clarified statement in [P3](#).
- Added links on rules.
- Added section heading for 5.7 [Migrating from 2.0 to 3.0](#)
- Moved text from the end of 4.3 [Higher-Level Protocols](#) to a new section 5.8 [Conversion to Plain Text](#)
- Rephrased the relationship between clauses [HL1](#) and [HL3](#) and [HL4](#) and [HL5](#)

Revision 22 being a proposed update, only changes between versions 21 and 23 are noted here.

Revision 21

- **Reissued** for Unicode 5.2.0.
- Added Section 4.4 [Bidi Conformance Testing](#).
- Added BN to Rule [X6](#) (removing certain characters).
- Clarified examples in Rule [N1](#) (affecting characters next to EN or AN characters).
- Added to [HL6](#) the clause: Those with a resolved directionality of L and whose bidi class is R or AL.
- Clarified the text at the start of 3 [Basic Display Algorithm](#).
- Added Bidi_Class and Bidi_Mirroring_Glyph property names.
- Added clarifications to 3.3.2 [Explicit Levels and Directions](#), to [X6](#), and to [N2](#).
- Fixed typos in 3.4 [Reordering Resolved Levels](#).
- Added links on items in Table 4, and clarified [BN](#) there.
- Removed a note in [N1](#).

Revision 20 being a proposed update, only changes between versions 19 and 21 are

noted here.

Revision 19

- Updated for Version 5.1.0.
- Clarified [BD6](#).
- Made some examples more explicit.
- Added the common problem case of separators in Section 5.5.
- Added notes on security concerns with RLO and LRO, and the use of dir="ltr" or "rtl" with web pages.
- Fixed example under [N2](#).
- Fixed example in Section 3.3.4 [Resolving Neutral Types](#)
- Made last part of Section 5.5 [Usage](#) into a new Section 5.6 [Separating Punctuation Marks](#), changed the term "Separators" and added a note on stateful controls.

Revision 18 being a proposed update, only changes between versions 19 and 17 are noted here.

Revision 17

- This revision incorporates successive changes. The latest changes were on 2006-02-24.
- Modified [L4](#) and [HL6](#), in conjunction with proposed property change to Bidi_Mirrored ([PRI #80](#))
- Added note on U+ FD3E (()) ORNATE LEFT PARENTHESIS and U+ FD3F (()) ORNATE RIGHT PARENTHESIS.
- Used new format for conformance clause numbering.
- Added caution on use of higher-level protocols, after [UAX9-C2](#).
- Some wording changes in 6 [Mirroring](#), for consistency with new L4 and HL6.
- Moved text to [Shaping and line breaking](#), and added note on SHY in 3.4 there
- Removed two notes indicating that the conformance clauses override clause C13 of Unicode 4.0.
- Changed some references to Unicode4.0

Revision 16 being a proposed update, only changes between versions 17 and 15 are noted here.

Revision 15:

- Minor editing
- Fixed section Number for Mirroring
- Changed "Tracking Number" to Revision
- Added note on U+0CBF KANNADA VOWEL SIGN I
- Added note after N1, and clarified example after N2.
- Fixed references to sections of the Unicode Standard

Revision 14:

- Aliased directional run and level run
- Pointed to DerivedBidiClass.txt for unassigned character assignments.

Revision 13:

- [4. Bidirectional Conformance](#): added explicit clauses.
- [4.3. Higher-Level Protocols](#):
 - Added clarifying text, and renumbered options.
 - Removed option regarding number shaping (because it was irrelevant to bidirectional ordering).
 - Broadened the ability to override on the basis of context, and clarified number handling.
 - Made clear that bidi could be applied to segments
- [1. Introduction](#): added note that the changes in [4. Bidirectional Conformance](#) override clause C13 of Unicode 4.0 [[Unicode](#)], and tighten the conformance requirements from what they had been previously.
- Minor editing for clarification.

Revision 11:

- Updated for Unicode 4.0.
- Added note on [canonical equivalence](#)
- Added [Joiners](#) section on ZWJ and ZWNJ
- Clarified [L2](#) and examples following.
- Added a section on the interaction of [shaping](#) and bidirectional reordering.
- Moved lists for unassigned characters into UCD.html (also now explicit in DerivedBidiClass.txt)
- Updated references for Newline Guidelines (because the UAX is incorporated into the 4.0 book)
- The first two sections were rearranged, with [Reference Code](#) going into [Implementation Notes](#), and [Mirroring](#) in its own section at the end.
 - This is *not* highlighted in the proposed text.
- Sections were renumbered and the table of contents is more detailed.
 - This is *not* highlighted in the proposed text.
- Misc editing.

Revision 10:

- Updated for Unicode 3.2.
- Updated UAX boilerplate in the status section.

Revision 9:

- Clarified the language of [P2](#)

- Corrected the implementation note on “Retaining Format Codes” in [Implementation Notes](#)
 - Minor editing
-

Copyright © 2000-2012 Unicode, Inc. All Rights Reserved. The Unicode Consortium makes no expressed or implied warranty of any kind, and assumes no liability for errors or omissions. No liability is assumed for incidental and consequential damages in connection with or arising out of the use of the information or programs contained or accompanying this technical report. The Unicode [Terms of Use](#) apply.

Unicode and the Unicode logo are trademarks of Unicode, Inc., and are registered in some jurisdictions.