Technical Reports

L2/23-042

Proposed Update Unicode® Standard Annex #9

UNICODE BIDIRECTIONAL ALGORITHM

Version	Unicode 15.1.0 (draft 2)
Editors	Mark Davis (mark@unicode.org), Ken Whistler (ken@unicode.org)
Date	2022-11-18
This Version	https://www.unicode.org/reports/tr9/tr9-47.html
Previous Version	https://www.unicode.org/reports/tr9/tr9-46.html
Latest Version	https://www.unicode.org/reports/tr9/
Latest Proposed Update	https://www.unicode.org/reports/tr9/proposed.html
Revision	47

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*This is a **draft** document which may be updated, replaced, or superseded by other documents at any time.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

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 formatting characters for special circumstances. 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 characters 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 characters 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.

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. With the exception of the directional formatting characters, the remaining bidirectional types and characters are called neutral. The algorithm uses the implicit bidirectional types of the characters in a text to arrive at a reasonable display ordering for text.

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 Characters

Three types of explicit directional formatting characters are used to modify the standard implicit Unicode Bidirectional Algorithm (UBA). In addition, there are implicit directional formatting characters, the *right-to-left* and *left-to-right* marks. The effects of all of these formatting characters are limited to the current paragraph; thus, they are terminated by a *paragraph separator*.

These formatting characters all have the property *Bidi_Control*, and are divided into three groups:

Implicit Directional Formatting Characters	LRM, RLM, ALM
Explicit Directional Embedding and Override Formatting Characters	LRE, RLE, LRO, RLO, PDF
Explicit Directional Isolate Formatting Characters	LRI, RLI, FSI, PDI

Although the term *embedding* is used for some explicit formatting characters, the text within the scope of the embedding formatting characters 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 formatting characters, however. Characters within an isolate cannot affect the ordering of characters outside it, or vice versa. The effect that an isolate as a whole has on the ordering of the surrounding characters is the same as that of a neutral character, whereas an embedding or override roughly has the effect of a strong character.

Directional isolate characters were introduced in Unicode 6.3 after it became apparent that directional embeddings usually have too strong an effect on their surroundings and are thus unnecessarily difficult to use. The new characters were introduced instead of changing the behavior of the existing ones because doing so might have had an undesirable effect on those existing documents that do rely on the old behavior. Nevertheless, the use of the directional isolates instead of embeddings is encouraged in new documents – once target platforms are known to support them.

On web pages, the *explicit* directional formatting characters (of all types – embedding, override, and isolate) should be replaced by other mechanisms suitable for HTML and CSS. For information on the correspondence between explicit directional formatting characters and equivalent HTML5 markup and CSS properties, see *Section 2.7, Markup and Formatting Characters*.

2.1 Explicit Directional Embeddings

The following characters 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, that phrase could be marked as being embedded right-to-left text. Embeddings can be nested one inside another, and in isolates and overrides.

Abbr.	Code Point	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. (PDF will be described in *Section 2.3, Terminating Explicit Directional Embeddings and Overrides*.)

2.2 Explicit Directional Overrides

The following characters allow the bidirectional character types to be overridden when required for special cases, such as for part numbers. They are to be avoided wherever

possible, because of security concerns. For more information, see [UTR36]. Directional overrides can be nested one inside another, and in embeddings and isolates.

Abbr.	Code Point	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 characters 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 character terminates the scope of the last LRE, RLE, LRO, or RLO whose scope has not yet been terminated.

Abbr.	Code Point	Name	Description
PDF	U+202C	POP DIRECTIONAL FORMATTING	End the scope of the last LRE, RLE, RLO, or LRO.

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

2.4 Explicit Directional Isolates

The following characters signal that a piece of text is to be treated as directionally isolated from its surroundings. They are very similar to the explicit embedding formatting characters. However, 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, and is assigned the corresponding display position in the surrounding text. Furthermore, the text inside the isolate has no effect on the ordering of the text outside it, and vice versa.

In addition to allowing the embedding of strongly directional text without unduly affecting the bidirectional order of its surroundings, one of the isolate formatting characters also offers an extra feature: embedding text while inferring its direction heuristically from its constituent characters.

Isolates can be nested one inside another, and in embeddings and overrides.

Abbr.	Code Point	Name	Description
LRI	U+2066	LEFT-TO-RIGHT ISOLATE	Treat the following text as isolated and left-to-right.
RLI	U+2067	RIGHT-TO-LEFT ISOLATE	Treat the following text as isolated and right-to-left.
FSI	U+2068	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 characters will be made clear in the discussion of the algorithm.

2.5 Terminating Explicit Directional Isolates

The following character terminates the scope of the last LRI, RLI, or FSI whose scope has not yet been terminated, as well as the scopes of any subsequent LREs, RLEs, LROs, or RLOs whose scopes have not yet been terminated.

Abbr.	Code Point	Name	Description
PDI	U+2069	POP DIRECTIONAL ISOLATE	End the scope of the last LRI, RLI, or FSI.

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

2.6 Implicit Directional Marks

These characters are very light-weight formatting. 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 Point	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 non-Arabic character
ALM	U+061C	ARABIC LETTER MARK	Right-to-left zero-width Arabic 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.

2.7 Markup and Formatting Characters

The explicit formatting characters introduce state into the plain text, which must be maintained when editing or displaying the text. Processes that are modifying the text without being aware of this state may inadvertently affect the rendering of large portions of the text, for example by removing a PDF.

The Unicode Bidirectional Algorithm is designed so that the use of explicit formatting characters can be equivalently represented by out-of-line information, such as stylesheet information or markup. Conflicts can arise if markup and explicitly formatting characters are both used in the same paragraph. Where available, markup should be used instead of the explicit formatting characters: for more information, see [UnicodeXML]. However, any alternative representation is only to be defined by reference to the behavior of the corresponding explicit formatting characters in this algorithm, to ensure conformance with the Unicode Standard.

HTML5 [HTML5] and CSS3 [CSS3Writing] provide support for bidi markup as follows:

Unicode	Equivalent Markup	Equivalent CSS	Comment
RLI ... PDI	dir = "rtl"	direction:rtl; unicode-bidi:isolate	dir attribute on any element
LRI ... PDI	dir = "ltr"	direction:ltr; unicode-bidi:isolate	dir attribute on any element
FSI ... PDI	<bdi>, dir = "auto"	unicode-bidi:plaintext	dir attribute on any element
RLE ... PDF		direction:rtl; unicode-bidi:embed	markup not available in HTML
LRE ... PDF		direction:ltr; unicode-bidi:embed	markup not available in HTML
RLO ... PDF		direction:rtl; unicode-bidi:bidi-override	markup not available in HTML
LRO ... PDF		direction:ltr; unicode-bidi:bidi-override	markup not available in HTML
FSI RLO . . . PDF PDI	<bdo dir = "rtl">	direction:rtl; unicode-bidi:isolate-override	
FSI LRO . . . PDF PDI	<bdo dir = "ltr">	direction:ltr; unicode-bidi:isolate-override	

Unlike HTML4.0, HTML5 does not provide exact equivalents for LRE, RLE, LRO, and RLO, although the dir attribute and the BDO element as outlined above should in most cases work as well or better than those formatting characters. When absolutely necessary, CSS can be used to get exact equivalents for LRE, RLE, LRO, and RLO, as well as for LRI, RLI, and FSI.

Whenever plain text is produced from a document containing markup, the equivalent formatting characters should be introduced, so that the correct ordering is not lost. For example, whenever cut and paste results in plain text this transformation should occur.

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 bidirectional character types is initialized, with one entry for each character in the original text. The value of each entry is the Bidi_Class property value of the respective character. A list of embedding levels, with one level per character, is then initialized. Note that the original characters are referenced in *Section 3.3.5, Resolving Neutral and Isolate Formatting Types*.
- **Resolution of the embedding levels.** A series of rules is applied to the lists of embedding levels and bidirectional character types. Each rule operates on the current values of those lists, and can modify those values. The original characters and their Bidi_Paired_Bracket and Bidi_Paired_Bracket_Type property values are

referenced in the application of certain rules. The result of this phase is a modified list of embedding levels; the list of bidirectional 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

3.1.1 Basics

BD1. The *bidirectional character types* are values assigned to each Unicode character, including unassigned characters. The formal property name in the *Unicode Character Database* [UCD] is 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 125, a value referred to as *max_depth* in the rest of this document.

As rules X1 through X8 will specify, embedding levels are set by explicit formatting characters (embedding, isolate, and override); 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. A maximum explicit level of 125 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.

For implementation stability, this specification now guarantees that the value of 125 for max_depth will not be increased (or decreased) in future versions. Thus, it is safe for implementations to treat the max_depth value as a constant. The max_depth value has been 125 since UBA Version 6.3.0.

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 directional override status is set by using explicit directional formatting characters. 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)*.

As specified below, level runs are important at two different stages of the Bidirectional Algorithm. The first stage occurs after rules X1 through X9 have assigned an explicit embedding level to each character on the basis of the paragraph direction and the explicit directional formatting characters. At this stage, in rule X10, level runs are used to build up the units to which subsequent rules are applied. Those rules further adjust each character's embedding level on the basis of its implicit bidirectional type and those of other characters in the unit – but not outside it. The level runs resulting from these resolved embedding levels are then used in the actual reordering of the text by rule L2. The following example illustrates level runs at this later stage of the algorithm.

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

Paragraph level:   0

Resolved levels:   000000011111110000000000
```

Notice that the neutral character (space) between THE and CAR gets the level of the surrounding characters. The level of the neutral characters could be changed by inserting appropriate directional marks around neutral characters, or using explicit directional formatting characters.

3.1.2 Matching Explicit Directional Formatting Characters

BD8. An *isolate initiator* is a character of type LRI, RLI, or FSI.

As rules X5a through X5c will specify, an isolate initiator raises the embedding level for the characters following it when the rules enforcing the depth limit allow it.

BD9. The *matching PDI* for a given isolate initiator is the one determined by the following algorithm:

- Initialize a counter to one.
- Scan the text following the isolate initiator to the end of the paragraph while incrementing the counter at every isolate initiator, 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 given isolate initiator. Otherwise, there is no matching PDI for it.

Note that all formatting characters except for isolate initiators and PDIs are ignored when finding the matching PDI.

Note that this algorithm assigns a matching PDI (or lack of one) to an isolate initiator whether the isolate initiator raises the embedding level or is prevented from doing so by the depth limit rules.

As rule X6a will specify, a matching PDI returns the embedding level to the value it had before the isolate initiator that the PDI matches. The PDI itself is assigned the new embedding level. If it does not match any isolate initiator, or if the isolate initiator did not raise the embedding level, it leaves the embedding level unchanged. Thus, an isolate initiator and its matching PDI are always assigned the same explicit embedding level, which is the one outside the isolate. In the later stages of the Bidirectional Algorithm, an isolate initiator and its matching PDI function as invisible neutral characters, and their embedding level then helps ensure that the isolate has the effect of a neutral character on the display order of the text outside it, and is assigned the corresponding display position in the surrounding text.

BD10. An *embedding initiator* is a character of type LRE, RLE, LRO, or RLO.

Note that an embedding initiator initiates either a directional embedding or a directional override; its name omits overrides only for conciseness.

As rules X2 through X5 will specify, an embedding initiator raises the embedding level for the characters following it when the rules enforcing the depth limit allow it.

BD11. The *matching PDF* for a given embedding initiator is the one determined by the following algorithm:

- Initialize a counter to one.
- Scan the text following the embedding initiator:
  - At an isolate initiator, skip past the matching PDI, or if there is no matching PDI, to the end of the paragraph.
  - At the end of a paragraph, or at a PDI that matches an isolate initiator whose text location is before the embedding initiator's location, stop: the embedding initiator has no matching PDF.
  - At an embedding initiator, increment the counter.
  - At a PDF, decrement the counter. If its new value is zero, stop: this is the matching PDF.

Note that this algorithm assigns a matching PDF (or lack of one) to an embedding initiator whether it raises the embedding level or is prevented from doing so by the depth limit rules.

Although the algorithm above serves to give a precise meaning to the term “matching PDF”, note that the overall Bidirectional Algorithm never actually calls for its use to find the PDF matching an embedding initiator. Instead, rules X1 through X7 specify a mechanism that determines what embedding initiator scope, if any, is terminated by a PDF, i.e. which valid embedding initiator a PDF matches.

As rule X7 will specify, a matching PDF returns the embedding level to the value it had before the embedding initiator that the PDF matches. If it does not match any embedding initiator, or if the embedding initiator did not raise the embedding level, a PDF leaves the embedding level unchanged.

As rule X9 will specify, once explicit directional formatting characters have been used to assign embedding levels to the characters in a paragraph, embedding initiators and PDFs are removed (or virtually removed) from the paragraph. Thus, the embedding levels assigned to the embedding initiators and PDFs themselves are irrelevant. In this, embedding initiators and PDFs differ from isolate initiators and PDIs, which continue to play a part in determining the paragraph’s display order as mentioned above.

BD12. The *directional isolate status* is a Boolean value set by using isolate formatting characters: it is true when the current embedding level was started by an isolate initiator.

BD13. An *isolating run sequence* is a maximal sequence of level runs such that for all level runs except the last one in the sequence, the last character of the run is an isolate initiator whose matching PDI is the first character of the next level run in the sequence. It is maximal in the sense that if the first character of the first level run in the sequence is a PDI, it must not match any isolate initiator, and if the last character of the last level run in the sequence is an isolate initiator, it must not have a matching PDI.

The set of isolating run sequences in a paragraph can be computed by the following algorithm:

- Start with an empty set of isolating run sequences.
- For each level run in the paragraph whose first character is not a PDI, or is a PDI that does not match any isolate initiator:
  - Create a new level run sequence, and initialize it to contain just that level run.
  - While the level run currently last in the sequence ends with an isolate initiator that has a matching PDI, append the level run containing the matching PDI to the sequence. (Note that this matching PDI must be the first character of its level run.)
  - Add the resulting sequence of level runs to the set of isolating run sequences.

Note that:

- Each level run in a paragraph belongs to exactly one isolating run sequence.
- In the absence of isolate initiators, each isolating run sequence in a paragraph consists of exactly one level run, and each level run constitutes a separate isolating run sequence.
- For any two adjacent level runs in an isolating run sequence, since one ends with an isolate initiator whose matching PDI starts the other, the two must have the same embedding level. Thus, all the level runs in an isolating run sequence have the same embedding level.
- When an isolate initiator raises the embedding level, both the isolate initiator and its matching PDI, if any, get the original embedding level, not the raised one. Thus, if the matching PDI does not immediately follow the isolate initiator in the paragraph, the isolate initiator is the last character in its level run, but the matching PDI, if any, is the first character of its level run and immediately follows the isolate initiator in the same isolating run sequence. On the other hand, the level run following the isolate initiator in the paragraph starts a new isolating run sequence, and the level run preceding the matching PDI (if any) in the paragraph ends its isolating run sequence.

In the following examples, assume that:

- The paragraph embedding level is 0.
- No character sequence $text_i$ contains explicit formatting characters or paragraph separators.
- The dots are used only to improve the example's visual clarity; they are not part of the text.
- The characters in the paragraph text are assigned embedding levels as loosely described above such that they form the set of level runs given in each example.

Example 1

Paragraph text: $text_1 \cdot$ **RLE** $\cdot text_2 \cdot$ **PDF** $\cdot$ **RLE** $\cdot text_3 \cdot$ **PDF** $\cdot text_4$

Level runs:

- $\boxed{text_1}$ – level 0
- $\boxed{text_2 \cdot text_3}$ – level 1
- $\boxed{text_4}$ – level 0

Resulting isolating run sequences:

- $text_1$ – level 0
- $text_2 \cdot text_3$ – level 1
- $text_4$ – level 0

Example 2

Paragraph text: $text_1 \cdot \mathbf{RLI} \cdot text_2 \cdot \mathbf{PDI} \cdot \mathbf{RLI} \cdot text_3 \cdot \mathbf{PDI} \cdot text_4$

Level runs:

- $text_1 \cdot \mathbf{RLI}$ – level 0
- $text_2$ – level 1
- $\mathbf{PDI} \cdot \mathbf{RLI}$ – level 0
- $text_3$ – level 1
- $\mathbf{PDI} \cdot text_4$ – level 0

Resulting isolating run sequences:

- $text_1 \cdot \mathbf{RLI}$ $\mathbf{PDI} \cdot \mathbf{RLI}$ $\mathbf{PDI} \cdot text_4$ – level 0
- $text_2$ – level 1
- $text_3$ – level 1

Example 3

Paragraph text: $text_1 \cdot \mathbf{RLI} \cdot text_2 \cdot \mathbf{LRI} \cdot text_3 \cdot \mathbf{RLE} \cdot text_4 \cdot \mathbf{PDF} \cdot text_5 \cdot \mathbf{PDI} \cdot text_6 \cdot \mathbf{PDI} \cdot text_7$

Level runs:

- $text_1 \cdot \mathbf{RLI}$ – level 0
- $text_2 \cdot \mathbf{LRI}$ – level 1
- $text_3$ – level 2
- $text_4$ – level 3
- $text_5$ – level 2
- $\mathbf{PDI} \cdot text_6$ – level 1
- $\mathbf{PDI} \cdot text_7$ – level 0

Resulting isolating run sequences:

- $text_1 \cdot \mathbf{RLI}$ $\mathbf{PDI} \cdot text_7$ – level 0
- $text_2 \cdot \mathbf{LRI}$ $\mathbf{PDI} \cdot text_6$ – level 1
- $text_3$ – level 2
- $text_4$ – level 3
- $text_5$ – level 2

As rule X10 will specify, an isolating run sequence is the unit to which the rules following it are applied, and the last character of one level run in the sequence is considered to be immediately followed by the first character of the next level run in the sequence during this

phase of the algorithm. Since those rules are based on the characters' implicit bidirectional types, an isolate really does have the same effect on the ordering of the text surrounding it as a neutral character – or, to be more precise, a pair of neutral characters, the isolate initiator and the PDI, which behave in those rules just like neutral characters.

3.1.3 Paired Brackets

The following definitions utilize the normative properties Bidi_Paired_Bracket and Bidi_Paired_Bracket_Type defined in the `BidiBrackets.txt` file [Data9] of the *Unicode Character Database* [UCD].

BD14. An *opening paired bracket* is a character whose Bidi_Paired_Bracket_Type property value is Open and whose current bidirectional character type is ON.

BD15. A *closing paired bracket* is a character whose Bidi_Paired_Bracket_Type property value is Close and whose current bidirectional character type is ON.

BD16. A *bracket pair* is a pair of characters consisting of an opening paired bracket and a closing paired bracket such that the Bidi_Paired_Bracket property value of the former or its canonical equivalent equals the latter or its canonical equivalent and which are algorithmically identified at specific text positions within an isolating run sequence. The following algorithm identifies all of the bracket pairs in a given isolating run sequence:

- Create a fixed-size stack for exactly 63 elements each consisting of a bracket character and a text position. Initialize it to empty.
- Create a list for elements each consisting of two text positions, one for an opening paired bracket and the other for a corresponding closing paired bracket. Initialize it to empty.
- Inspect each character in the isolating run sequence in logical order.
    - If an opening paired bracket is found and there is room in the stack, push its Bidi_Paired_Bracket property value and its text position onto the stack.
    - If an opening paired bracket is found and there is no room in the stack, stop processing BD16 for the remainder of the isolating run sequence.
    - If a closing paired bracket is found, do the following:
        1. Declare a variable that holds a reference to the current stack element and initialize it with the top element of the stack.
        2. Compare the closing paired bracket being inspected or its canonical equivalent to the bracket in the current stack element.
        3. If the values match, meaning the two characters form a bracket pair, then
            - Append the text position in the current stack element together with the text position of the closing paired bracket to the list.
            - Pop the stack through the current stack element inclusively.
        4. Else, if the current stack element is not at the bottom of the stack, advance it to the next element deeper in the stack and go back to step 2.
        5. Else, continue with inspecting the next character without popping the stack.
- Sort the list of pairs of text positions in ascending order based on the text position of the opening paired bracket.

Note that bracket pairs can only occur in an isolating run sequence because they are processed in rule N0 after explicit level resolution. See *Section 3.3.2, Explicit Levels and Directions*.

Examples of bracket pairs

```
Text                    Pairings
1 2 3 4 5 6 7 8
a ) b ( c               None
a ( b ] c               None
a ( b ) c               2-4
a ( b [ c ) d ]         2-6
a ( b ] c ) d           2-6
a ( b ) c ) d           2-4
a ( b ( c ) d           4-6
a ( b ( c ) d )         2-8, 4-6
a ( b { c } d )         2-8, 4-6
```

3.1.4 Additional Abbreviations

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
NI	Neutral or Isolate formatting character (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).
o	The text ordering type (L or R) that matches the direction **opposite** the embedding level direction (even or odd). Note that o is the opposite of e.
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 I 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 9.2, 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 DerivedBidiClass.txt [DerivedBIDI] 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 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	ALM, 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, NO-BREAK SPACE, ...
	NSM	Nonspacing Mark	Characters with the General_Category values: Mn (Nonspacing_Mark) and Me (Enclosing_Mark)
	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 bidirectional character types, explicit formatting characters, and bracket pairs to produce a list of resolved levels. This resolution process consists of the following steps:

- Applying rule P1 to split the text into paragraphs, and for each of these:
  - Applying rules P2 and P3 to determine the paragraph level.
  - Applying rule X1 (which employs rules X2–X8) to determine explicit embedding levels and directions.
  - Applying rule X9 to remove many control characters from further consideration.
  - Applying rule X10 to split the paragraph into isolating run sequences and for each of these:
    - Applying rules W1–W7 to resolve weak types.
    - Applying rules N0–N2 to resolve neutral types.
    - Applying rules I1–I2 to resolve implicit embedding levels.

3.3.1 The Paragraph Level

P1. *Split the text into separate paragraphs. A paragraph separator (type B) 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 isolate initiator and its matching PDI or, if it has no matching PDI, the end of the paragraph.*

Note that:

- Because paragraph separators delimit text in this algorithm, the character found by this rule will generally be the first strong character after a paragraph separator or at the very beginning of the text.
- The characters between an isolate initiator and its matching PDI are ignored by this rule because a directional isolate is supposed to have the same effect on the ordering of the surrounding text as a neutral character, and the rule ignores neutral characters.
- The characters between an isolate initiator and its matching PDI are ignored by this rule even if the depth limit (as defined in rules X5a through X5c below) prevents the isolate initiator from raising the embedding level. This is meant to make the rule easier to implement.
- Embedding initiators (but not the characters within the embedding) are ignored in this rule.

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 explicit directional formatting characters (embedding, override, and isolate), by applying the explicit level rule X1. This performs a logical pass over the paragraph, applying rules X2–X8 to each characters in turn. The following variables are used during this pass:

- A *directional status stack* of max_depth+2 entries where each entry consists of:
  - An embedding level, which is at least zero and at most max_depth.
  - A directional override status.
  - A directional isolate status.

  In addition to supporting the usual destructive “pop” operation, the stack also allows read access to its last (i.e. top) entry without popping it. For efficiency, that last entry can be kept in a separate variable instead of on the directional status stack, but it is easier to explain the algorithm without that optimization. At the start of the pass, the directional status stack is initialized to an entry reflecting the paragraph embedding level, with the directional override status neutral and the directional isolate status false; this entry is not popped off until the end of the paragraph. During the pass, the directional status stack always contains entries for all the directional embeddings, overrides, and isolates within which the current position lies – except those that would overflow the depth limit – in addition to the paragraph level entry at the start of the stack. The last entry reflects the innermost valid scope within which the pass's current position lies. Implementers may find it useful to include more information in each stack entry. For example, in an isolate entry, the location of the isolate initiator

could be used to create a mapping from the location of each valid isolate initiator to the location of the matching PDI, or vice versa. However, such optimizations are beyond the scope of this specification.

- A counter called the *overflow isolate count*.
  This reflects the number of isolate initiators that were encountered in the pass so far without encountering their matching PDIs, but were invalidated by the depth limit and thus are not reflected in the directional status stack. They are nested one within the other and the stack's last scope. This count is used to determine whether a newly encountered PDI matches and terminates the scope of an overflow isolate initiator, thus decrementing the count, as opposed to possibly matching and terminating the scope of a valid isolate initiator, which should result in popping its entry off the directional status stack. It is also used to determine whether a newly encountered PDF falls within the scope of an overflow isolate initiator and can thus be completely ignored (regardless of whether it matches an embedding initiator within the same overflow isolate or nothing at all).
- A counter called the *overflow embedding count*.
  This reflects the number of embedding initiators that were encountered in the pass so far without encountering their matching PDF, or encountering the PDI of an isolate within which they are nested, but were invalidated by the depth limit, and thus are not reflected in the directional status stack. They are nested one within the other and the stack's last scope. This count is used to determine whether a newly encountered PDF matches and terminates the scope of an overflow embedding initiator, thus decrementing the count, as opposed to possibly matching and terminating the scope of a valid embedding initiator, which should result in popping its entry off the directional status stack. However, this count does *not* include embedding initiators encountered within the scope of an overflow isolate (i.e. encountered when the overflow isolate count above is greater than zero). The scopes of those overflow embedding initiators fall within the scope of an overflow isolate and are terminated when the overflow isolate count turns zero. Thus, they do not need to be counted. In fact, if they were counted in the overflow embedding count, there would be no way to properly update that count when a PDI matching an overflow isolate initiator is encountered: without a stack of the overflow scopes, there would be no way to know how many (if any) overflow embedding initiators fall within the scope of that overflow isolate.
- A counter called the *valid isolate count*.
  This reflects the number of isolate initiators that were encountered in the pass so far without encountering their matching PDIs, and have been judged valid by the depth limit, i.e. all the entries on the stack with a true directional isolate status. It ignores all embeddings and overrides, and is used to determine without having to look through the directional status stack whether a PDI encountered by the pass when the overflow isolate count is zero matches some valid isolate initiator or nothing at all. A PDI encountered when this counter is above zero terminates the scope of the isolate initiator it matches, as well as the embeddings and overrides nested within it – which appear above it on the stack, or are reflected in the overflow embedding count.

Note that there is no need for a valid embedding count in order to tell whether a PDF encountered by the pass matches a valid embedding initiator or nothing at all. That can be decided by checking the directional isolate status of the last entry on the directional status stack and the number of entries on the stack. If the last entry has a true directional isolate status, it is for a directional isolate within whose scope the PDF lies. Since the PDF cannot match an embedding initiator outside that isolate, and there are no embedding entries within the isolate, it matches nothing at all. And if the last entry has a false directional

isolate status, but is also the only entry on the stack, it belongs to paragraph level, and thus once again the PDF matches nothing at all.

As each character is processed, these variables' values are modified and the character's explicit embedding level is set as defined by rules X2 through X8 on the basis of the character's bidirectional type and the variables' current values.

X1. *At the beginning of a paragraph, perform the following steps:*

- *Set the stack to empty.*
- *Push onto the stack an entry consisting of the paragraph embedding level, a* ***neutral*** *directional override status, and a* ***false*** *directional isolate status.*
- *Set the overflow isolate count to zero.*
- *Set the overflow embedding count to zero.*
- *Set the valid isolate count to zero.*
- *Process each character iteratively, applying rules X2 through X8. Only embedding levels from 0 through max_depth are valid in this phase.* (Note that in the resolution of levels in rules I1 and I2, the maximum embedding level of max_depth+1 can be reached.)

Explicit Embeddings

X2. *With each RLE, perform the following steps:*

- *Compute the least* ***odd*** *embedding level greater than the embedding level of the last entry on the directional status stack.*
- *If this new level would be valid, and the overflow isolate count and overflow embedding count are both zero, then this RLE is valid. Push an entry consisting of the new embedding level,* ***neutral*** *directional override status, and* ***false*** *directional isolate status onto the directional status stack.*
- *Otherwise, this is an overflow RLE. If the overflow isolate count is zero, increment the overflow embedding count by one. Leave all other variables unchanged.*

For example, assuming the overflow counts are both zero, level 0 → 1; levels 1, 2 → 3; levels 3, 4 → 5; and so on. At max_depth or if either overflow count is non-zero, the level remains the same (overflow RLE).

X3. *With each LRE, perform the following steps:*

- *Compute the least* ***even*** *embedding level greater than the embedding level of the last entry on the directional status stack.*
- *If this new level would be valid, and the overflow isolate count and overflow embedding count are both zero, then this LRE is valid. Push an entry consisting of the new embedding level,* ***neutral*** *directional override status, and* ***false*** *directional isolate status onto the directional status stack.*
- *Otherwise, this is an overflow LRE. If the overflow isolate count is zero, increment the overflow embedding count by one. Leave all other variables unchanged.*

For example, assuming the overflow counts are both zero, levels 0, 1 → 2; levels 2, 3 → 4; levels 4, 5 → 6; and so on. At max_depth or max_depth-1 (which, being even, would have to go to max_depth+1) or if either overflow count is non-zero, the level remains the same (overflow LRE).

Explicit Overrides

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

X4. *With each RLO, perform the following steps:*

- *Compute the least **odd** embedding level greater than the embedding level of the last entry on the directional status stack.*
- *If this new level would be valid, and the overflow isolate count and overflow embedding count are both zero, then this RLO is valid. Push an entry consisting of the new embedding level, **right-to-left** directional override status, and **false** directional isolate status onto the directional status stack.*
- *Otherwise, this is an overflow RLO. If the overflow isolate count is zero, increment the overflow embedding count by one. Leave all other variables unchanged.*

X5. *With each LRO, perform the following steps:*

- *Compute the least **even** embedding level greater than the embedding level of the last entry on the directional status stack.*
- *If this new level would be valid, and the overflow isolate count and overflow embedding count are both zero, then this LRO is valid. Push an entry consisting of the new embedding level, **left-to-right** directional override status, and **false** directional isolate status onto the directional status stack.*
- *Otherwise, this is an overflow LRO. If the overflow isolate count is zero, increment the overflow embedding count by one. Leave all other variables unchanged.*

Isolates

X5a. *With each RLI, perform the following steps:*

- *Set the RLI's embedding level to the embedding level of the last entry on the directional status stack.*
- *If the directional override status of the last entry on the directional status stack is not neutral, reset the current character type from RLI to L if the override status is left-to-right, and to R if the override status is right-to-left.*
- *Compute the least **odd** embedding level greater than the embedding level of the last entry on the directional status stack.*
- *If this new level would be valid and the overflow isolate count and the overflow embedding count are both zero, then this RLI is valid. Increment the valid isolate count by one, and push an entry consisting of the new embedding level, **neutral** directional override status, and **true** directional isolate status onto the directional status stack.*
- *Otherwise, this is an overflow RLI. Increment the overflow isolate count by one, and leave all other variables unchanged.*

X5b. *With each LRI, perform the following steps:*

- *Set the LRI's embedding level to the embedding level of the last entry on the directional status stack.*

- *If the directional override status of the last entry on the directional status stack is not neutral, reset the current character type from LRI to L if the override status is left-to-right, and to R if the override status is right-to-left.*
- *Compute the least **even** embedding level greater than the embedding level of the last entry on the directional status stack.*
- *If this new level would be valid and the overflow isolate count and the overflow embedding count are both zero, then this LRI is valid. Increment the valid isolate count by one, and push an entry consisting of the new embedding level, **neutral** directional override status, and **true** directional isolate status onto the directional status stack.*
- *Otherwise, this is an overflow LRI. Increment the overflow 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 new embedding level is not set 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 B, BN, RLE, LRE, RLO, LRO, PDF, RLI, LRI, FSI, and PDI:*

- *Set the current character's embedding level to the embedding level of the last entry on the directional status stack.*
- *Whenever the directional override status of the last entry on the directional status stack is not neutral, reset the current character type according to the directional override status of the last entry on the directional status stack.*

In other words, if the directional override status of the last entry on the directional status stack is neutral, then characters retain their normal types: Arabic characters stay AL, Latin characters stay L, spaces stay WS, and so on. If the directional override status is right-to-left, then characters become R. If the directional override status is left-to-right, then characters become L.

Note that the current embedding level is not changed by this rule.

Terminating Isolates

A PDI terminates the scope of the isolate initiator it matches. It also terminates the scopes of all embedding initiators within the scope of the matched isolate initiator for which a matching PDF has not been encountered. If it does not match any isolate initiator, it is ignored.

***X6a.** With each PDI, perform the following steps:*

- *If the overflow isolate count is greater than zero, this PDI matches an overflow isolate initiator. Decrement the overflow isolate count by one.*
- *Otherwise, if the valid isolate count is zero, this PDI does not match any isolate initiator, valid or overflow. Do nothing.*
- *Otherwise, this PDI matches a valid isolate initiator. Perform the following steps:*

    - *Reset the overflow embedding count to zero.* (This terminates the scope of those overflow embedding initiators within the scope of the matched isolate initiator whose scopes have not been terminated by a matching PDF, and which thus lack a matching PDF.)
    - *While the directional isolate status of the last entry on the stack is false, pop the last entry from the directional status stack.* (This terminates the scope of those valid embedding initiators within the scope of the matched isolate initiator whose scopes have not been terminated by a matching PDF, and which thus lack a matching PDF. Given that the valid isolate count is non-zero, the directional status stack before this step is executed must contain an entry with directional isolate status true, and thus after this step is executed the last entry on the stack will indeed have a true directional isolate status, i.e. represent the scope of the matched isolate initiator. This cannot be the stack's first entry, which always belongs to the paragraph level and has a false directional status, so there is at least one more entry below it on the stack.)
    - *Pop the last entry from the directional status stack and decrement the valid isolate count by one.* (This terminates the scope of the matched isolate initiator. Since the preceding step left the stack with at least two entries, this pop does not leave the stack empty.)
- *In all cases, look up the last entry on the directional status stack left after the steps above and:*
    - *Set the PDI's level to the entry's embedding level.*
    - *If the entry's directional override status is not neutral, reset the current character type from PDI to L if the override status is left-to-right, and to R if the override status is right-to-left.*

Note that the level assigned to an isolate initiator is always the same as that assigned to the matching PDI.

Terminating Embeddings and Overrides

A PDF terminates the scope of the embedding initiator it matches. If it does not match any embedding initiator, it is ignored.

X7. *With each PDF, perform the following steps:*

- *If the overflow isolate count is greater than zero, do nothing.* (This PDF is within the scope of an overflow isolate initiator. It either matches and terminates the scope of an overflow embedding initiator within that overflow isolate, or does not match any embedding initiator.)
- *Otherwise, if the overflow embedding count is greater than zero, decrement it by one.* (This PDF matches and terminates the scope of an overflow embedding initiator that is not within the scope of an overflow isolate initiator.)
- *Otherwise, if the directional isolate status of the last entry on the directional status stack is false, and the directional status stack contains at least two entries, pop the last entry from the directional status stack.* (This PDF matches and terminates the scope of a valid embedding initiator. Since the stack has at least two entries, this pop does not leave the stack empty.)
- *Otherwise, do nothing.* (This PDF does not match any embedding initiator.)

End of Paragraph

X8. *All explicit directional embeddings, overrides and isolates are completely terminated at the end of each paragraph.*

- Explicit paragraph separators (bidirectional character type B) indicate the end of a paragraph. As such, they are **not** included in any embedding, override or isolate. They are simply assigned the paragraph embedding level.

3.3.3 Preparations for Implicit Processing

The explicit embedding levels that have been assigned to the characters by the preceding rules will soon be further adjusted on the basis of the characters' implicit bidirectional types. The adjustment made for a given character will then depend on the characters around it. However, this dependency is limited by logically dividing the paragraph into sub-units, and doing the subsequent implicit processing on each unit independently.

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

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

  See Section 5, *Implementation Notes*, for information on implementing the algorithm without removing the formatting characters.

- 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 6.1, *Joiners*.
- Note that FSI, LRI, RLI, and PDI characters are **not** removed. As indicated by the rules below, they are used, in part, to determine the paragraph's isolating run sequences, within which they are then treated as neutral characters. Nevertheless, they are of course zero-width characters and, like LRM and RLM, should not be visible in the final output.

X10. *Perform the following steps:*

- *Compute the set of isolating run sequences as specified by BD13, based on the bidirectional types of the characters and the embedding levels assigned by the rules above (X1–X9).*
- *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:*
  - *For sos, compare the level of the first character in the sequence with the level of the character preceding it in the paragraph (not counting characters removed by X9), 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 (not counting characters removed by X9), and if there is none or the last character of the sequence is an isolate initiator (lacking a matching PDI), with the paragraph embedding level.*
  - *If the higher level is odd, the sos or eos is R; otherwise, it is L.*
  - Note that these computations must use the embedding levels assigned by the rules above, before any changes are made to them in the steps below.

- *Apply rules W1–W7, N0–N2, and I1–I2, in the order in which they appear below, to each of the isolating run sequences, applying one rule to all the characters in the sequence in the order in which they occur in the sequence before applying another rule to any part of the sequence. The order that one isolating run sequence is treated relative to another does not matter. When applying a rule to an isolating run sequence, the last character of each level run in the isolating run sequence is treated as if it were immediately followed by the first character in the next level run in the sequence, if any.*

Here are some examples, each of which is assumed to be a paragraph with base level 0 where no character sequence $text_i$ contains explicit directional formatting characters or paragraph separators. The dots in the examples are intended to separate elements for visual clarity; they are not part of the text.

Example 1: $text_1$ · **RLE** · $text_2$ · **LRE** · $text_3$ · **PDF** · $text_4$ · **PDF** · **RLE** · $text_5$ · **PDF** · $text_6$

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: $text_1$ · **RLI** · $text_2$ · **LRI** · $text_3$ · **PDI** · $text_4$ · **PDI** · **RLI** · $text_5$ · **PDI** · $text_6$

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: $text_1$ · **RLE** · $text_2$ · **LRI** · $text_3$ · **RLE** · $text_4$ · **PDI** · $text_5$ · **PDF** · $text_6$

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

3.3.4 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.

First, each nonspacing mark is resolved based on the character it follows.

W1. *Examine each nonspacing mark (NSM) in the isolating run sequence, and change the type of the NSM to Other Neutral if the previous character is an isolate initiator or PDI, and to the type of the previous character otherwise. If the NSM is at the start of the isolating run sequence, it will get the type of* ***sos****.* (Note that in an isolating run sequence, an isolate initiator followed by an NSM or any type other than PDI must be an overflow isolate initiator.)

Assume in this example that *sos* is R:

```
AL  NSM NSM → AL  AL  AL

sos NSM     → sos R

LRI NSM     → LRI ON

PDI NSM     → PDI ON
```

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* ***sos****) is found. If an AL is found, change the type of the European number to Arabic number.*

```
AL EN     → AL AN

AL NI EN  → AL NI AN

sos NI EN → sos NI EN

L NI EN   → L NI EN

R NI EN   → R NI 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* ***sos****) is found. If an L is found, then change the type of the European number to L.*

```
L  NI EN → L  NI  L

R  NI EN → R  NI  EN
```

3.3.5 Resolving Neutral and Isolate Formatting Types

In the next phase, neutral and isolate formatting (i.e. NI) characters are resolved one isolating run sequence at a time. Its results are that all NIs become either **R** or **L**. Generally, NIs take on the direction of the surrounding text. In case of a conflict, they take on the embedding direction. 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.

Bracket pairs within an isolating run sequence are processed as units so that both the opening and the closing paired bracket in a pair resolve to the same direction. Note that this rule is applied based on the current bidirectional character type of each paired bracket and not the original type, as this could have changed under X6. The current bidirectional character type may also have changed under a previous iteration of the *for* loop in N0 in the case of nested bracket pairs.

N0. *Process bracket pairs in an isolating run sequence sequentially in the logical order of the text positions of the opening paired brackets using the logic given below. Within this scope, bidirectional types EN and AN are treated as R.*

- *Identify the bracket pairs in the current isolating run sequence according to BD16.*
- *For each bracket-pair element in the list of pairs of text positions*
    a. *Inspect the bidirectional types of the characters enclosed within the bracket pair.*
    b. *If any strong type (either L or R) matching the embedding direction is found, set the type for both brackets in the pair to match the embedding direction.*

```
o [ e ] o → o e e e o

o [ o e ] → o e o e e
```

```
o [ NI e ] → o e NI e e
```

c. *Otherwise, if there is a strong type it must be opposite the embedding direction. Therefore, test for an established context with a preceding strong type by checking backwards before the opening paired bracket until the first strong type (L, R, or sos) is found.*

1. *If the preceding strong type is also opposite the embedding direction, context is established, so set the type for both brackets in the pair to that direction.*

```
o [ o ] e → o o o o e

o [ o NI ] o → o o o NI o o
```

2. *Otherwise set the type for both brackets in the pair to the embedding direction.*

```
e [ o ] o → e e o e o

e [ o ] e → e e o e e
```

d. *Otherwise, there are no strong types within the bracket pair. Therefore, do not set the type for that bracket pair.*

```
e ( NI ) o → e ( NI ) o
```

Note that if the enclosed text contains no strong types the bracket pairs will both resolve to the same level when resolved individually using rules N1 and N2.

- *Any number of characters that had original bidirectional character type NSM prior to the application of W1 that immediately follow a paired bracket which changed to L or R under N0 should change to match the type of their preceding bracket.*

Example 1. Bracket pairs are resolved sequentially in logical order of the opening paired brackets.

(RTL paragraph direction)

Storage	AB	(	CD	[	&	ef	]	!	)	gh
Bidi_Class	R	ON	R	ON	ON	L	ON	ON	ON	L
N0 applied (first pair)		N0b: ON→R							N0b: ON→R	
N0 applied (second pair)				N0c2: ON→R			N0c2: ON→R			
Display	gh(![ef&]DC)BA									

Example 2. Bracket pairs enclosing mixed strong types take the paragraph direction.

(RTL paragraph direction)

Storage	smith		(	fabrikam		ARABIC	)		HEBREW
Bidi_Class	L	WS	ON	L	WS	R	ON	WS	R
N0 applied			N0b: ON→R				N0b: ON→R		
Display	WERBEH (CIBARA fabrikam) smith								

Note that in the above example, the resolution of the bracket pairs is stable if the order of smith and HEBREW, or fabrikam and ARABIC, is reversed.

Example 3. Bracket pairs enclosing strong types opposite the embedding direction with additional strong-type context take the direction opposite the embedding direction.

(RTL paragraph direction)

Storage	ARABIC		book	(	s	)
Bidi_Class	R	WS	L	ON	L	ON
N0 applied				N0c1: ON→L		N0c1: ON→L
Display	book(s) CIBARA					

N1. *A sequence of NIs 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 NIs. The start-of-sequence (**sos**) and end-of-sequence (**eos**) types are used at isolating run sequence boundaries.*

```
 L  NI   L  →   L  L   L
 R  NI   R  →   R  R   R
 R  NI  AN  →   R  R  AN
 R  NI  EN  →   R  R  EN
AN  NI   R  →  AN  R   R
AN  NI  AN  →  AN  R  AN
AN  NI  EN  →  AN  R  EN
EN  NI   R  →  EN  R   R
EN  NI  AN  →  EN  R  AN
EN  NI  EN  →  EN  R  EN
```

N2. *Any remaining NIs take the embedding direction.*

```
NI → e
```

The embedding direction for the given NI 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   NI eos → L   L eos

R   NI eos → R   e eos

sos NI L   → sos e L

sos NI 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 3.3.4, *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
```

3.3.6 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 level max_depth+1 as a result of this process.) This results in the following rules:

I1. *For all characters with an even (left-to-right) embedding level, 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 level, 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 formatting characters (FSI, LRI, RLI, and PDI) preceding a segment separator or paragraph separator, and*
4. *Any sequence of whitespace characters and/or isolate formatting characters (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 is shown in the "Storage" row in the example tables. The invisible, zero-width formatting characters LRI, RLI, and PDI are represented with the symbols **>**, **<**, and **=**, respectively. The application of the rules from *Section 3.3, Resolving Embedding Levels* and of the Rule L1 results in the resolved levels listed in the "Resolved Levels" row. (Since these examples only make use of the isolate formatting characters, Rule X9 does not remove any characters. Note that Example 3 would not work if it used embeddings instead because the two right-to-left phrases would have merged into a single right-to-left run, together with the neutral punctuation in between.) Each successive row thereafter shows 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, second, and third examples is 0 (left-to-right direction), and for the fourth example is 1 (right-to-left direction).

Example 1. (embedding level = 0)

Storage	car means CAR.
Resolved levels	00000000001110
Reverse level 1	car means RAC.
Display	car means RAC.

Example 2. (embedding level = 0)

Storage	<car MEANS CAR.=
Resolved levels	0222111111111110
Reverse level 2	<rac MEANS CAR.=
Reverse levels 1-2	<.RAC SNAEM car=
Display	.RAC SNAEM car

Example 3. (embedding level = 0)

Storage	he said “<car MEANS CAR=.” “<IT DOES=,” she agreed.
Resolved levels	00000000002221111111111000001111111000000000000000
Reverse level 2	he said “<rac MEANS CAR=.” “<IT DOES=,” she agreed.
Reverse levels 1-2	he said “<RAC SNAEM car=.” “<SEOD TI=,” she agreed.
Display	he said “RAC SNAEM car.” “SEOD TI,” she agreed.

Example 4. (embedding level = 1)

Storage	DID YOU SAY ’>he said “<car MEANS CAR=”=‘?
Resolved levels	11111111111111222222222244433333333322111
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
Display	?‘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 Yes.*

- *The Bidi_Mirrored property is defined by Section 4.7, Bidi Mirrored 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 9.2, Arabic* of [Unicode]). Shaping is logically applied *after* Rule I2 of the Bidirectional Algorithm and is limited to characters within the same level run. (Note that there is no practical difference between limiting shaping to a level run and an isolating run sequence because the isolate initiator and PDI characters are defined to have joining type U, i.e. non-joining. Thus, the characters before and after a directional isolate will not join across the isolate, even if the isolate is empty or overflows the depth limit.) 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 062C JEEM	ع 0639 AIN	ل 0644 LAM	م 0645 MEEM
L	**L**	**R**	**R**

One can use explicit directional formatting characters 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"/"**rtl**">LRO *JEEM AIN* PDF RLO *LAM MEEM* PDF</p>
- <p dir="ltr"/"**rtl**"><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 JEEM-F	ع AIN-I	م MEEM-F	ﻟ LAM-I	م MEEM-F	ﻟ LAM-I	ج JEEM-F	ع 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

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.

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 formatting characters) in the order described by Section 3, Basic Display Algorithm, of this annex. In particular, this includes definitions BD1–BD16 and steps P1–P3, X1–X10, W1–W7, N0–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.

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.*

> **Note:** The use of higher-level protocols introduces interchange problems, since the text may be displayed differently as plain text; see Section 6.5, *Conversion to Plain Text*. This can have security implications. Higher-level protocols are recommended wherever the semantics of segment order are more significant than those of displayed order, as is the case for source text. For detailed examples for which use of HL4 would be recommended, see Section 4.3.1, *HL4 Example1 for XML* and Section 4.3.2, *HL4 Example2 for Program Text*. For more information, see Section 4.1, *Bidirectional Ordering*, in Unicode Technical Standard #55, “Unicode Source Code Handling” [UTS55], as well as Unicode Technical Report #36, “Unicode Security Considerations” [UTR36].

4.1 Boundary Neutrals

The goal in marking a formatting 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 formatting 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 Characters

As with any Unicode characters, systems do not have to support any particular explicit directional formatting character (although it is not generally useful to include a terminating character 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 ALM, RLM and LRM are supported.
- *Non-isolate bidirectionality.* The implicit Bidirectional Algorithm, the implicit directional marks, and the explicit non-isolate directional formatting characters are supported: ALM, RLM, LRM, LRE, RLE, LRO, RLO, PDF.
- *Full bidirectionality.* The implicit Bidirectional Algorithm, the implicit directional marks, and all the explicit directional formatting characters are supported: ALM, 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. This does* ***not*** *apply when deciding how to treat FSI in rule X5c.*

- 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 characters.*

- 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 directional formatting characters as defined in the algorithm were inserted into the text. For example, a style sheet or markup can modify 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 7, *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 bidirectional type 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.

4.3.1 HL4 Example 1 for XML

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.

4.3.2 HL4 Example 2 for Program Text

Consider the following two lines:

(1) `x + tav == 1`

(2) `x + 1 == תו`

Internally, they are the same except that the ASCII identifier `tav` in line (1) is replaced by the Hebrew identifier `תו` in line (2). However, with a plain text display (with left-to-right paragraph direction) the user will be misled, thinking that line (2) is a comparison between `(x + 1)` and `תו`, whereas it is actually a comparison between `(x + תו)` and `1`. The misleading rendering of (2) occurs because the directionality of the identifier `תו` influences subsequent weakly-directional tokens, so that the entire sequence "`1 == תו`" is at a higher resolved level.

This is illustrated in the first row of the following table, wherein characters at a resolved level higher than the embedding level are highlighted. Note that while the RTL display of that expression (second row) is not misleading, as the left-to-right directionality of `x` does not influence the subsequent text, a similar issue would arise if the terms were swapped (third row).

Paragraph direction	Underlying Representation											Display
LTR	x		+		ת	ו		=	=		1	x + 1 == תו
RTL	x		+		ת	ו		=	=		1	1 == תו + x
RTL	ת	ו		+		x		=	=		1	x == 1 + תו

It would be better to apply protocol HL4 when displaying these expressions, treating each identifier as a separate segment, thus isolating it from the rest of the source text, and then ordering the segments in a consistent direction, as shown in the following table.

Segment order	Segments									Display
LTR	x		+		תו		==		1	x + תו == 1
RTL	x		+		תו		==		1	1 == תו + x
RTL	תו		+		x		==		1	1 == x + תו

4.4 Bidirectional Conformance Testing

The *Unicode Character Database* [UCD] includes two files that provide conformance tests for implementations of the Bidirectional Algorithm [Tests9]. One of the test files, `BidiTest.txt`, comprises exhaustive test sequences of bidirectional types up to a given length, currently 4. The other test file, `BidiCharacterTest.txt`, contains test sequences of

explicit code points, including, for example, bracket pairs. The format of each test file is described in the header of that file.

5 Implementation Notes

5.1 Reference Code

Reference implementations of the Bidirectional Algorithm written in C and in Java are available. The source code can be downloaded from [Code9]. Implementers are encouraged to use these resources to test their implementations.

The reference code is designed to follow the steps of the algorithm without applying any optimizations. An example of an effective optimization is to first test for right-to-left characters and invoke the Bidirectional Algorithm only if they are present. Another example of optimization is in matching bracket pairs. The bidirectional bracket pairs (the characters with Bidi_Paired_Bracket_Type property values Open and Close) constitute a subset of the characters with bidirectional type ON. Conversely, the characters with a bidirectional type distinct from ON have the Bidi_Paired_Bracket_Type property value None. Therefore, lookup of Bidi_Paired_Bracket_Type property values for the identification of bracket pairs can be optimized by restricting the processing to characters whose bidirectional type is ON.

An online demo is also available at [Demo9], which shows the results of the Bidirectional Algorithm, as well as the embedding levels and the rules invoked for each character. Implementers are cautioned when using that online demo that it implements the rules for UBA as of Version 6.2, and has not been updated for the major changes to UBA in Version 6.3 and subsequent versions. The online demo also does not handle supplemental characters gracefully.

5.2 Retaining BNs and Explicit Formatting Characters

Some implementations may wish to retain the explicit directional embedding and override formatting characters and BNs when running the algorithm. In fact, retention of these formatting characters and BNs is important to users who need to display a graphic representation of hidden characters, and who thus need to obtain their visual positions for display.

The following describes how this may be done by implementations that do retain these characters through the steps of the algorithm. Note that this description 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 rules X2 through X5, insert an initial step setting the explicit embedding or override character's embedding level to the embedding level of the last entry on the directional status stack. This applies to RLE, LRE, RLO, and LRO.
- In rule X6, remove the exclusion of BN characters. In other words, apply the rule to all types except B, RLE, LRE, RLO, LRO, PDF, RLI, LRI, FSI, and PDI.
- In rule X7, add a final step setting the embedding level of the PDF to the embedding level of the last entry on the directional status stack, in all cases.
- In rule X9, do not remove any characters, but turn all RLE, LRE, RLO, LRO, and PDF characters into BN.
- In rule X10, when determining the sos and eos for an isolating run sequence, skip over any BNs when looking for the character preceding the isolating run sequence's

first character and following its last character.

- In rule W1, search backward from each NSM to the first character in the isolating run sequence whose bidirectional type is not BN, and set the NSM to ON if it is an isolate initiator or PDI, and to its type otherwise. If the NSM is the first non-BN character, change the NSM to 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 N0–N2, treat BNs that adjoin neutrals the same as those neutrals.
- In rules I1 and I2, ignore BN.
- In rule L1, include the embedding and override formatting characters and BNs together with whitespace characters and isolate formatting 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.

6 Usage

6.1 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.

6.2 Vertical Text

In the case of vertical line orientation, there are multiple ways to display bidirectional text. Some methods use the Bidirectional Algorithm, and some do not. The Unicode Standard does not specify whether text is presented with horizontal or vertical layout, or for vertical layout whether elements within the line are rotated. That is left up to higher-level protocols. For example, one of the common approaches for vertical line orientation is to rotate all the glyphs uniformly 90° clockwise. The Bidirectional Algorithm is used with this method. While some characters end up ordered from bottom to top, this method can represent a mixture of Arabic and Latin glyphs in the same way as occurs for horizontal line orientation.

Another possible approach is to render the text in a uniform single direction from top to bottom. This method has multiple variations to determine the orientation of characters. One variant uses the Bidirectional Algorithm to determine the level of the text, but then the levels are not used to reorder the text. Instead, the levels are used to determine the *rotation* of each segment 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 those characters are also ordered from top to bottom.

6.3 Formatting

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 "RLI I NEED WATER! PDI", and expired.

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

However, an often 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. This is the best approach when manually editing text or programmatically generating text meant to be edited, or dealing with an application that simply does not support explicit formatting characters.

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 explicit formatting characters, which can easily get out of sync if not fully supported by editors and other string manipulation. Nevertheless, the explicit formatting characters are absolutely necessary in cases where text of one direction contains text of the opposite direction which itself contains text of the original direction. Such cases are not as rare as one might think, because Latin-script brand names, technical terms, and abbreviations are often written in their original Latin characters when used in non-Latin-script text, including right-to-left text, as in the following:

Memory: it is called "RLI AN INTRODUCTION TO java PDI" - $19.95 in hardcover.

Display: it is called "java OT NOITCUDORTNI NA" - $19.95 in hardcover.

Thus, when text is programmatically generated by inserting data into a template, and is not intended for later manual editing, and a particular insert happens to be of the opposite direction to the template's text, it is easiest to wrap the insert in explicit formatting

characters (or their markup equivalent) declaring its direction, without analyzing whether it is really necessary to do so, or if the job could be done just with stateless directional marks.

Furthermore, in this common scenario, it is highly recommended to use directional isolate formatting characters as opposed to directional embedding formatting characters (once targeted display platforms are known to support isolates). This is because embeddings affect the surrounding text similarly to a strong character, whereas directional isolates have the effect of a neutral. The embeddings' stronger effect is often difficult to anticipate and is rarely useful. To demonstrate, here is the example above with embeddings instead of isolates:

Memory: it is called "RLEAN INTRODUCTION TO javaPDF" - $19.95 in hardcover.

Display: it is called "$19.95 - "java OT NOITCUDORTNI NA in hardcover.

This, of course, is not the intended display, and is due to the number “sticking” to the preceding RTL embedding (along with all the neutral characters in between), just as it would “stick” to a preceding RTL character.

Directional isolates also offer a solution to the very common case where the direction of the text to be programmatically inserted is not known. Instead of analyzing the characters of the text to be inserted in order to decide whether to use an LRE or RLE (or LRI or RLI - or nothing at all), the software can take the easy way out and *always* wrap each unknown-direction insert in an FSI and PDI. Thus, an FSI instead of an RLI in the example above would produce the same display. FSI's first-strong heuristic is not infallible, but it will work most of the time even on mixed-script text.

Although wrapping inserts in isolates is a useful technique, it is best not to wrap text that is known to contain no opposite-direction characters that are not already wrapped in an isolate. Unnecessary layers of wrapping not only add bulk and complexity; they can also wind up exceeding the depth limit and rendering ineffective the innermost isolates, which can make the text display incorrectly. One very common case of an insert that does not need wrapping is one known to be localized to the context locale, e.g. a translated message with all its inserted values either themselves localized, or wrapped in an isolate.

6.4 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 explicit formatting characters (LRE, RLE, and PDF or LRI, RLI, FSI, and PDI) 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 explicit formatting characters 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 explicit formatting characters when converting to plain text, but that is not generally supported.)

6.5 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 characters 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 characters. If the same text were converted to HTML4.0 the attribute dir = "rtl" would be added to the paragraph element.

For program text, whose proper display is subject to higher-level protocols, such a conversion to plain text needs to be performed in a way that does not change the semantics of the program. It is recommended that computer languages allow for the insertion of some formatting characters in appropriate locations without changing the meaning of a program; for computer languages that allow this insertion, a procedure is specified for conversion to plain text. See *Section 4.1, Whitespace*, in *Unicode Standard Annex #31, Identifiers and Syntax* [UAX31], and *Section 5.2, Conversion to Plain Text*, in *Unicode Technical Standard #55, Unicode Source Code Handling* [UTS55].

7 Mirroring

The mirrored property is important to ensure that the correct characters 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 is used to express the intended semantic, 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 or 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 BidiMirroring.txt 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.

Migration Issues

There are two major enhancements in the Unicode 6.3 version of the UBA:

- Directional isolates
- Bracket Pairs

Implementations of the new directional isolates should see very few compatibility issues; the UBA has been carefully modified to minimize differences for older text written without them. There are a few edge cases near the limit of the number of levels where there are some differences, but those are not likely to be encountered in practice.

With bracket pairs, there may be more changes. The problem is that without knowing (or having good UI access to) the directional marks or embeddings, people have constructed text with the correct visual appearance but incorrect underlying structure (eg …[…[…, appearing as …[…]…). The new algorithm catches cases like these, because such malformed sequences of brackets are not matched.

However, there are some cases where older implementations without rule N0 produced the desired appearance, and newer implementations will not. The user feedback on implementations was sufficiently positive that the decision was made to add N0.

There are also incompatibilities from some implementation's failing to updating correctly to previous versions of Unicode, notably in the mishandling solidus such that "T 1/2" (T is an Arabic character) appears incorrectly as "2/1 T".

To mitigate compatibility problems, it is strongly recommended that implementations take the following steps:

- Add appropriate directional formatting characters on both any parentheses that are resolved with rule N0 so that they appear properly on older systems. This can be done with directional marks (RLM or LRM) on both sides of each parenthesis. For forward compatibility, text authored on older systems should use semantically correct brackets (with directional formatting characters as necessary) to ensure correct display on systems with implementations after Unicode 6.3.
- Add the appropriate explicit embedding around any sequence of numbers + solidus + numbers.

Section Reorganization

In Unicode 6.3, there was significant reorganization of the text. The following table shows the new and old section numbers.

Unicode 6.3	Unicode 6.2
2.4 Explicit Directional Isolates	n/a
2.5 Terminating Explicit Directional Isolates	n/a
2.6 Implicit Directional Marks	2.4
3.3.3 Preparations for Implicit Processing	n/a
3.3.4 Resolving Weak Types …3.3.6 Resolving Implicit Levels	3.3.3 …3.3.5
6.1 Joiners	5.3
6.2 Vertical Text	5.4
6.3 Formatting	5.5
6.4 Separating Punctuation Marks	5.6
6.5 Conversion to Plain Text	n/a
Migration Issues	5.7

Acknowledgments

Mark Davis created the initial version of this annex and maintains the text. Aharon Lanin and Andrew Glass made substantial additions to Revision 29 (Unicode 6.3.0). Robin Leroy made substantial additions to Revision 46 (Unicode 15.0.0).

Thanks to the following people for their contributions to the Bidirectional Algorithm or for their feedback on earlier versions of this annex: Ahmed Talaat (أحمد طلعت), Alaa Ghoneim (علاء غنيم), Asmus Freytag, Avery Bishop, Ayman Aldahleh (أيمن الدحلة), Behdad Esfahbod (بهداد اسفهبد), Doug Felt, Dwayne Robinson, Eric Mader, Ernest Cline, Gidi Shalom-Bendor (גידי שלום-בן דור), Gilead Almosnino (גלעד אלמוסנינו), Isai Scheinberg, Israel Gidali (ישראל גידלי), Joe Becker, John McConnell, Jonathan Kew, Jonathan Rosenne (יונתן רוזן), Kamal Mansour (كمال منصور), Kenneth Whistler, Khaled Sherif (خالد شريف), Koji Ishii, Laurențiu Iancu, Maha Hassan (مها حسن), Markus Scherer, Martin Dürst, Mati Allouche (מתתיהו אלוש), Michel Suignard, Mike Ksar (ميشيل قصار), Murray Sargent, Paul Nelson, Pedro Navarro, Peter Constable, Rick McGowan, Robert Steen, Roozbeh Pournader (روزبه پورنادر), Solra Bizna, Steve Atkin, and Thomas Milo (تُومَاسْ مِيلُو).

References

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

Modifications

The following summarizes modifications from the previous version of this annex.

Revision 47

- **Proposed Update** for Unicode 15.1.0
- TBD

~~**Revision 46**~~

- ~~**Reissued** for Unicode 15.0.0~~
- ~~Amended the text of the note under UAX9-C2 to recommend use of higher-level protocols in special circumstances such as source text.~~
- ~~Added an extended example of use of HL4 for program text in Section 4.3.2, *HL4 Example 2 for Program Text*.~~

Previous revisions can be accessed with the “Previous Version” link in the header.
