Appendix A: Directionality

The Unicode standard always uses logical backing store order. When text is presented in horizontal lines, most scripts display characters from left to right. However, there are several scripts where the natural ordering of horizontal text is from right to left (such as Arabic or Hebrew). If all of the text has the same horizontal direction, then the ordering of the display text is unambiguous. However, when there is bidirectional text (a mixture of left-to-right and right-to-left horizontal text) there is a fundamental ambiguity in the ordering of the displayed characters.

The following describes the algorithm used to perform reordering for bidirectional Unicode text. It extends the implicit model currently in use on a number of computers, and combines it with explicit controls for special circumstances. In most cases, there is no need to include additional information with the text to get correct display ordering. However, additional information can be included in the text when necessary by means of a small set of directional formatting codes.

In principle, the Unicode standard does not supply formatting codes; formatting is left up to higher-level protocols. However, in the case of bidirectional text, there are certain circumstances where an implicit bidirectional ordering is not sufficient to produce comprehensible text. To deal with these cases, a minimal set of directional formatting codes are supplied to control the ordering of characters when rendered. This allows exact control of the display ordering for legible interchange, and also allows vendors who still want plain text for simple items like filenames or labels to ensure that that text can always be correctly ordered for display.

The directional formatting codes are *only* used to influence the display ordering of text. In all other respects they are ignored: they should have no effect on the comparison of text, nor on word breaks, parsing, or numeric analysis. In addition, these codes are to be completely ignored when text is presented in vertical lines. In that case, the default order is for all characters to proceed from top to bottom (except for non-spacing accents, which have the same position relative to the base character). The choice of whether text is to be presented horizontally or vertically is left up to higher-level protocols.

The ordering of bidirectional text depends upon the directional properties of the text. The Character Properties section following lists the ranges of characters that have each particular directional character type.

NOTE: 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 the Arabic block description for more details on naming digits.

Directional Ordering Codes

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

Explicit Directional Embedding

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

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

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

Explicit directional overrides

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

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

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

Terminating Explicit Directional Code

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

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

Implicit Directional Marks

These characters are very light weight codes. They act exactly like right-to-left or left-to-right characters, except that they do not display (or have any other semantic effect). Their use is often more convenient than the explicit embeddings or overrides, since their scope is much more local (as will be made clear following).

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

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

Basic Display Algorithm

This algorithm may be coded differently for speed, but logically speaking follows two phases. The input is a stream of text, up to a *block separator* (such as a paragraph separator), and the character types for each character.

Phase 1. Resolution of the embedding levels of the text. In this phase, the directional character types, plus the explicit controls, are used to produce resolved embedding levels.

Phase 2. Reordering the text on a line-by-line basis, using the resolved embedding levels.

Embedding levels are numbers that indicate the embedding level of text. ("Embedding levels" in this text are determined both by override controls and by embedding controls.) Odd levels are right-to-left, and even levels are left-to-right. The minimum embedding level of text is zero, and the maximum depth is level 15. (The reason for having a limitation is to provide a precise stack limit for implementations to guarantee the same results. Fifteen levels is far more than sufficient for ordering: the display becomes rather muddied with more than a small number of embeddings.)

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 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.

Example. In 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), while lowercase letters stand for left-to-right characters (such as English or Russian).

Backing store:	`foo is FOO BAR in arabic`
Character types:	`LLL-LL-RRR-RRR-LL-LLLLLL`
Resolved levels:	`000000011111110000000000`

Notice that the neutral character (space) between FOO and BAR gets the level of the surrounding characters. This is how the implicit directional marks have an effect; by inserting appropriate directional marks around neutral characters, the level of the neutral characters can be changed.

Resolving Embedding Levels

Combining character types and explicit codes to produce a list of resolved levels lies at the heart of the bidirectional algorithm. This process consists of seven steps: determining the base level; determining explicit embedding levels and directions; determining explicit overrides; determining embedding and override terminations; resolving numbers; resolving neutrals; and resolving implicit embedding levels.

The Base Level

First, determine the *base embedding level*, which determines the default horizontal orientation of the text in the current block.

B1. *In the text, find the first strong directional character, RLO or LRO. (Because block separators delimit text in this algorithm, this will generally be the first strong character after a block separator or at the very beginning of the text.)*

B2. *If the first strong directional character in the text is right-to-left or RLO, then set the base level to one, otherwise set it to zero.*

The direction of the base embedding level is called the *base direction* or *block direction.*

Explicit Levels & Directions

All explicit embedding levels are determined from the embedding and override codes. The directional level indicates how deeply the text is embedded, and the basic directional flow of the text. Each even level is a left-to-right embedding, and each odd level is a right-to-left embedding. Only levels from 0 to 15 are valid.

E1. *Begin at the base embedding level. Set the directional override status to neutral.*

E2. *With each RLE, remember (push) the current embedding level and override status. Reset the current level to the least greater odd level (if it would be valid), and the override status to neutral.*

For example, level 0 → 1; levels 1, 2 → 3; levels 3, 4 → 5;...13, 14 → 15; above 14, no change (don't change levels with LRE if the new level would be invalid).

E3. *With each LRE, remember (push) the current level and override status. Reset the current level to the least greater even level (if it would be valid), and the override status to neutral.*

For example, levels 0, 1 → 2; levels 2, 3 → 4; levels 4, 5 → 6; ...12, 13 → 14; above 13, no change (don't change levels with LRE if the new level would be invalid).

Explicit Overrides

A directional override changes all of the following characters within the current explicit embedding level to a given value, and sets the embedding level as with the embedding codes.

O1. *With each RLO, remember (push) the current override status and embedding level. Reset the current override status to be right-to-left, and the current level to the least greater odd level (if it would be valid).*

O2. *With each LRO, remember (push) the current override status and embedding level. Reset the current override status to be left-to-right, and the current level to the least greater even level (if it would be valid.).*

O3. *Whenever the directional override status is not neutral, reset the current character type to the directional override status.*

Resetting levels works as described for embeddings in the previous section.

For example, if the directional override status is neutral, then all intermediate characters retain their normal values: Arabic characters stay R, Latin characters stay L, neutrals stay N, etc.. If the directional override status is R, then all characters become R.

Terminating Embeddings & Overrides

There is a single code to terminate the scope of the current explicit code, whether an embedding or a directional override. All codes and pushed states are completely popped at block separators.

T4. *With each PDF, restore (pop) the last remembered (pushed) bidirectional state (embedding level and directional override). If there is no pushed state, ignore PDF.*

T5. *All explicit directional embeddings and overrides are completely terminated at Block separators. Return to the state as of E1.*

All overrides and resolution of numbers and neutrals take effect within the bounds of an embedding. That is, nothing within an embedding or override will effect the character direction of codes outside of that embedding, and vice versa. The one exception is in resolving neutrals (see N5 below).

Resolving Numbers

The text is parsed for numbers. This parsing does *not* span explicit directional controls: LRO, RLO, LRE, RLE, PDF, RLM, LRM. This pass will change the directional types of the character types European Number Separator, European Number Terminator, and Common Number Separator to be European Number text, Arabic Number text, andOther 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. All parsing does not span explicit controls; that is, any explicit control such as an RLE will terminate a number parsing.

The table below lists the abbreviations used in the following examples.

ES	European Number Separator	AL	Arabic Letter
ET	European Number Terminator	LL	Latin Letter
CS	Common Number Separator	HL	Hebrew Letter
EN	European Number text	N	Neutral
AN	Arabic Number text	sot	start of text
ON	Other Neutral text	eot	end of text

P0. *Search backwards from each instance of a European number until the first strong character (or block boundary) is found. If a character is found before a block boundary, and if that character belongs to the Arabic block, then change the type of the European number to Arabic number:*

```
AL,EN       =>    AL,AN
AL,N,EN     =>    AL,N,AN
sot,EN      =>    sot,EN
LL,EN       =>    LL,EN
HL,EN       =>    HL,EN
```

P1. *Separators change to numbers when surrounded by appropriate numbers:*

```
EN,CS,EN    =>    EN,EN,EN
AN,CS,AN    =>    AN,AN,AN
```

P2. *Terminators change to numbers when adjacent to an appropriate number:*

```
EN,ET       =>    EN,EN
ET,EN       =>    EN,EN
```

P3. Otherwise, terminators change to Other Neutral:

```
L,ES,EN      =>    L,N,EN
EN,CS,AN     =>    EN,N,AN
...
```

Resolving Neutrals

The next phase resolves the direction of the neutrals. The results of this phase are that all neutrals become either R or L.

Generally, neutral characters take on the direction of the surrounding text. In case of a conflict, they take on the embedding level. End-of-text and start-of-text are treated as if there were a character of the embedding level at that position. In this phase, block separators, segment separators, whitespace and other neutrals act the same, and will be indicated by an "N" in the examples. "e" represents the text ordering type (either L or R) that matches the embedding level direction in the examples.

N1. A sequence of neutrals takes the direction of the surrounding strong text.

```
R N R        =>    R R R
L N L        =>    L L L
```

N2. Where there is a conflict in adjacent strong directions, a sequence of neutrals takes the global direction.

```
L N R        =>    L e R
R N L        =>    R e L
```

Since end-of-text and start-of-text are treated as if they were characters of the embedding level at that position, the following examples are covered by this rule:

```
L N eot      =>    L e eot
R N eot      =>    R e eot
sot N L      =>    sot e L
sot N R      =>    sot e R
```

N3. Otherwise, in any sequence of neutrals and numbers, the neutrals go by the surrounding characters, as in the following rules. The numbers (whether European or Arabic) remain numeric.

```
R N EN N R   =>    R R EN R R
R N EN N L   =>    R R EN e L
L N EN N R   =>    L L EN e R
L N EN N L   =>    L L EN L L
R N AN N R   =>    R R AN R R
R N AN N L   =>    R R AN e L
L N AN N R   =>    L e AN R R
L N AN N L   =>    L L AN L L
```

N4. *When processing adjacent neutrals, any embedded text will be treated as if it were a single strong character of the appropriate direction. The following examples illustrate the effects on neutrals.*

```
R N [LRO <text> PDF] N L  =>  R R LRO <text> PDF L L
R N [RLE <text> PDF] N L  =>  R R RLE <text> PDF R L
```

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".
Visual:     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 since they are not surrounded on both sides (see number parsing).

However, if there is an adjacent left-to-right sequence, then Western numbers will adopt that direction:

```
Storage:    he said "IT IS A bmw 500, OK".
Visual:     he said "KO ,bmw 500 A SI TI".
```

Resolving Implicit Levels

In the final phase, the embedding level of text may be increased, based upon the resolved character type. Right-to-left text will always have an odd level, and left-to-right and numeric text will always have an even level. In addition, numeric text will always have a higher level than the base level, except in one special case. This results in the following rules:

I1. *If the global direction is even (left-to-right) then the right-to-left text goes up one level. Numeric text (AN) goes up two levels. Numeric text (EN) goes up two levels unless preceeded by left-to-right text.*

I2. *If the global direction is odd (right-to-left) then the left-to-right text and numeric text (EN or AN) goes up one level.*

The following table summarizes the results of the implicit algorithm. The "L" indicates a preceding character type.

Embedding Level (EL)	*Sequence Type*	*Result*
Even	L	EL
	R	EL+1
	AN	EL+2
	EN	EL+2
	(L) EN	EL
Odd	R	EL
	L	EL+1
	AN	EL+1
	EN	EL+1

Reordering Resolved Levels

The following describes the logical process of finding the correct display order. As before, this logical process is *not necessarily* the actual implementation, which may diverge for efficiency. As opposed to resolution phases, the following algorithm acts on a per-line basis.

L1. *Reset the embedding level of segment separators and trailing white space (including block separators) to be the base embedding level.*

In combination with the following, this means that trailing white space will appear at the visual end of the line (in the base direction). Tabulation will always have a consistant direction within a directional block.

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

This reverses a progressively larger series of substrings. The four examples below illustrate this.

Backing store:	`foo means FOO.`
Resolved levels:	`00000000001110`
Reverse level 1:	`foo means OOF.`
Backing store:	`foo MEANS FOO.`
Resolved levels:	`22211111111111`
Reverse level 2:	`oof MEANS FOO.`
Reverse levels 1,2:	`OOF SNAEM foo`
Backing store:	`he said "foo MEANS FOO."`
Resolved levels:	`00000000022211111111100`
Reverse level 2:	`he said "oof MEANS FOO."`
Reverse levels 1,2:	`he said "OOF SNAEM foo."`

Backing store:	`DID YOU SAY 'he said "foo MEANS FOO"'?`
Resolved levels:	`11111111111222222222244333333333221`
Reverse level 4:	`DID YOU SAY 'he said "oof MEANS FOO"'?`
Reverse levels 3,4:	`DID YOU SAY 'he said "OOF SNAEM foo"'?`
Reverse levels 2-4:	`DID YOU SAY '"oof MEANS FOO" dias eh'?`
Reverse levels 1-4:	`?'he said "OOF SNAEM foo"' YAS UOY DID`

NOTE: The correct appearance should be used for OPEN and CLOSE characters depending on their level: for example, OPEN PARENTHESIS appears as "(" when its resolved level is even, and as ")"when its resolved level is odd.

Conformance

The bidirectional algorithm specifies part of the intrinsic semantics of right-to-left characters. In the absence of a higher level protocol, systems that encode these characters must make use of the implicit bidirectional algorithm.

Explicit Formatting Codes

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

- No bidirectional formatting. No right-left characters are used.
- Implicit bidirectionality. The implicit bidirectional algorithm is present (including RLM and LRM).
- Full bidirectionality. Both the implicit bidirectional algorithm and the explicit directional formatting codes are included.

Examples of Higher-Level Protocols

The following are concrete examples of how systems may apply higher-level protocols to the ordering of bidirectional text.

- Override the basic level embedding (global direction). A higher-level protocol may provide for overriding the basic level embedding, either on a field, paragraph, document or system level.
- Override the number handling to provide for more (or less) sophisticated number parsing. For example, different types of numbers can be parsed differently; however, this requires additional information such as the language.
- Supplement or override the directional overrides or embedding codes by providing information via stylesheets about the embedding level or character direction.

- Remap the number shapes to match those of another set. For example, remap the Arabic number shapes to have the same appearance as the European numbers.

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

Usage

Because of the implicit character types and the heuristics for resolving neutral and numeric directional behavior, the implicit bidirectional ordering will generally produce the correct display without any further work. However, bad 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 bad case is that of neutrals on the boundary of an embedded language. This can be addressed by setting the level of the embedded streak correctly. For example, with all the text at level 0 the following occurs:

Backing store:	`he said "MEANS FOO!", and expired.`
Display result:	`he said "OOF SNAEM!", and expired.`

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

Display result:	`he said "!OOF SNAEM", and expired.`

Another method of doing this is to place a right directional mark after the exclamation mark. Since the exclamation mark is now not on a directional boundary, this produces the correct result.