Unicode Bidirectional Ordering Codes

First Draft	10/7/90	MED	
Second Draft	10/9/90	MED	Changed Numeric to Weak LR, added Weak RL
Third Draft	3/19/90	MED	Rolled in results of subcommittee work
Fourth Draft	3/19/90	MED	Cleared up ambiguities
Fifth Draft	3/26/90	MED	Simplified the number handling in response to subcommittee, added summary of proposals.

1. Introduction

As mentioned before, Unicode always uses logical backing-store order. When text is presented in horizontal lines, most languages display characters from left to right. However, there are several languages 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, whenever there is bidirectional text (a mixture of left-to-right and right-to-left horizontal text) there is a fundamental ambiguity in the ordering 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 Xerox and Apple computers, and combines it with explicit controls for special circumstances. In most cases, no additional information needs to be included 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, Unicode does not supply formatting codes, leaving that 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 counteract that, a minimal set of directional formatting codes are supplied to control the ordering of characters when presented on displays. This allows vendors who still want plain text as a base level for simple items like filenames or labels to ensure that that text can always be ordered correctly for display.

The directional formatting codes are 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, all characters 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.

2. Directional Character Types

The algorithm uses a specification of all Unicode characters as having one of eleven directional types:

Strong Types

L	Left-Right	English, most alphabetic, syllabic, logographic characters: Latin letters: U+0041..U+005A, U+0061..U+007A, U+00C0..U+00D6, U+00D8..U+00F6, U+00F8..U+00FF European..Modifier Letters: U+0100..U+02FF Greek..Armenian: U+0370..U+058F Ethiopian..Georgian: U+0700..U+10FF Hiragana..Han Compatibility: U+3040..U+3FFF Han: U+4000..<last assigned block> CZ Fullwidth ASCII..Fullwidth symbols: U+FF00..U+FFEF Super/Sub digits: U+2070, U+00B9, U+00B2..U+00B3, U+2074..U+2079, U+2080..U+2089 Roman Numerals: U+2160..U+2082 Left-Right Mark: U+200F
R	Right-Left	Arabic, Hebrew, & punctuation specific to those scripts: General Diacriticals: U+0300..U+036F Arabic and Hebrew: U+0590..U+06FF Arabic CZ Glyphs: U+FE80..U+FEFF Right-Left Mark: U+200F

Weak Types

W	Western Number	Western European digits: U+0030..U+0039 Farsi Digits: U+XXXX..U+XXXX Figure Space: U+2007
WS	Western Number Separator	Comma, Period: U+002C, U+002E
WT	Western Number Terminator	Plus sign: U+002B, Minus Sign: U+2212 Hyphen-Minus: U+002D Plus-Minus, Minus-Plus: U+YYYY Western Percents: U+0025, U+0030..U+0031 Degree, Minute, Second: U+YYYY...
A	Arabic Number	Arabic-based digits: U+0660..U+0669, U+066B..U+066F Arabic decimal & thousands separators: U+XXXX, U+XXXX
CS	Common Number Separator	Slash, Colon: U+002F, U+003A

Neutrals

B	Block Separator	Paragraph separator (PS) U+2029, Line separator (LS) U+2028, LF (U+000A), FF (U+000C), CR (U+000D)
S	Segment Separator	HT (U+0009)
WS	Whitespace	Space U+0020, NBSP U+00A0 General Punctuation Spaces: U+2000..U+200B
ON	Other Neutrals	All other characters: punctuation, symbols, unassigned

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.

The Character Properties section lists the ranges of characters that have each particular directional character type <<editor: copy the above information to there>>.

Since horizontal logographic characters are generally left-to-right, they have the Left-Right directional character type. When they need to be written from right-to-left, their direction can overridden as discussed below.

3. 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-left and left-right marks. All codes are limited to the current directional block; that is, their effects are terminated by a Block Separator.

1. Explicit Directional Embedding

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

LRE	Left-Right Embedding	Treat the following text as embedded left-to-right.
RLE	Right-Left Embedding	Treat the following text as embedded right-to-left.
PDE	Pop Directional Embedding	Restore the embedding level to what it was before the last LRE or RLE.

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..RDE.

2. Explicit Directional Overrides

The following codes allow the character types to be overridden in special cases, such as for part numbers.

RLO	Right-Left Override	Force following characters to be treated as strong right-to-left characters.
LRO	Left-Right Override	Force following characters to be treated as strong left-to-right characters.
SDO	Stop Directional Override	Stop overriding text.

The precise meaning of these codes will be made clear in the discussion of the algorithm. The right-left override, for example, can be used to force Left-Right text (such as English letters) to be written from right-to-left.

3. Implicit Directional Marks

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

These characters are very light-weight codes. They act exactly like right-left or left-right codes, 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 below).

There is no special mention of the directional neutral modifiers 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.

4. 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 (e.g. 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 produced resolved embedding levels.

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

Embedding levels are numbers that indicate the embedding level of text. Odd levels are left-right, and even levels are right-left. The minimum embedding level of text is zero, and the maximum level is 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, etc. 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.

In the examples given below, case is used to indicate different implicit character types for transmissibility. Upper-case letters stand for right-to-left characters (e.g. Arabic or Hebrew), while lower-case letters stand for left-to-right characters (e.g. English or Russian).

Example:

```
Backing store:       foo is FOO BAR in arabic
Character types:     LLL-LL-RRR-RRR-LL-LLLLLL
Resolved levels:     00000001111111000000000
```

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

5. Resolving Embedding Levels

In these phases, the character types and explicit codes are combined to produce a list of resolved levels. Logically speaking, this proceeds by resolving the following: the base level, explicit embedding levels, explicit directional types, resolved character types (numbers & neutrals), and resolved implicit embedding levels.

1. The Base Level

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

B1. Find the first strong directional character in the text

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-left, 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*.

2. Explicit Embeddings

All explicit embedding levels are determined from the embedding 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-right embedding, and each odd level is a right-left embedding. Only levels from 0 to 15 are valid.

E1. Begin at the base embedding level.

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

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 RLE if the new level would be invalid).

E3. With each LRE, remember the current level, and reset the current level to the least greater *even* ***level (if it would be valid).***

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 RLE if the new level would be invalid).

E4. With each RDE, restore the last remembered level. If there is none, ignore RDE.

E5. All directional embeddings are terminated at Block separators.

All overrides, and resolution of numbers and neutrals take effect within the bounds of an embedding. That is, nothing within an embedding will effect the character direction of codes outside of that embedding, and vice versa.

3. Explicit Overrides

A directional override changes all of the following characters within the current explicit embedding level to a given value.

O1. Directional overrides are terminated by an SDO, and also terminate at: any other explicit directional formatting code (RLE, LRE, PDE, RLO, LRO), a Block Separator, or End of text.

O2. Within the scope of a right directional override, all characters directional types are overriden to be right-left.

O2. Within the scope of a left directional override, all characters directional types are overriden to be left-right.

4. Resolving Numbers

Between any embedding codes, the text is parsed for numbers. This pass will change the directional types of characters of the following types:

WS	Western Number Separator
WT	Western Number Terminator
CS	Common Number Separator

to be either:

W	Western Number text
A	Arabic Number text
ON	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. All parsing does not span explicit controls: that is, any explicit control such as an RLE will terminate a number parsing.

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

```
W,WS,W      =>      W,W,W
W,CS,W      =>      W,W,W
A,CS,A      =>      A,A,A
```

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

```
W,WT        =>      W,W
WT,W        =>      W,W
```

P3. Otherwise, terminators change to Other Neutral:

```
L,WS,W      =>      L,N,W
W,CS,A      =>      W,N,A
...
```

5. 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. In some cases, Western numbers may also become 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. Also, the letter 'e' is a shorthand for representing the text ordering type (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.

```
E.g.
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.

```
E.g.
L N R      =>         L e L
```

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

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

[Retracted!

When numbers are mixed in, the situation is more complex.

N3. Any sequence of Western number text adjacent to Left-right text becomes Left-right. Any sequence of intervening neutrals are also converted.

```
E.g.
W L  =>      L L
W N L =>     L L L
W N W N L    =>     L L L L L
]
```

N4. Otherwise, in any sequence of neutrals and numbers, the neutrals go by the surrounding strong characters, as in N1 & N2.

The numbers (whether Western or Arabic) remain numeric. E.g.

```
R n W n R  =>         R R W R R
R n W n L  =>         R e L L L
L n W n R  =>         L L L e R
L n W n L  =>         L L L L L
R n A n R  =>         R R A R R
R n A n L  =>         R e A e L
```

```
L n A n R =>        L e A e R
L n A n L =>        L L A L L
```

N5. Any embedded level will be ignored when processing neutrals.

This means that the resolution of neutrals will be exactly the same if the embedded text were removed. Let the letter 'E' represents text which is explicitely embedded. The following examples illustrate the lack of effects on neutrals.

```
R n E n R =>        R R E R R
L n E n L =>        L L E L 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 (see above under 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".
```

6. Resolving Implicit Levels

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

I1. If the global direction is even (left-to-right) then the right-left text goes up one level, and the numeric text (W or A) goes up two levels.

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

The following table summarizes the results of the implicit algorithm.

Embedding Level(EL)	Sequence Type	Result
Even	L	EL
	R	EL+1
	A,W	EL+2
Odd	R	EL
	L,A,W	EL+1

6. Reordering Resolved Levels

The following describes the logical process of finding the correct display order. 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. Set 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. For example:

Example 1:

```
Backing store:       foo means FOO.
Resolved levels:     00000000001110
Reverse level 1:     foo means OOF.
```

Example 2:

```
Backing store:       foo MEANS FOO.
Resolved levels:     22211111111111
Reverse level 2:     oof MEANS FOO.
Reverse levels 1,2:  .OOF SNAEM foo
```

Example 3:

```
Backing store:       he said"foo MEANS FOO."
Resolved levels:     000000000222111111111100
Reverse level 2:     he said "oof MEANS FOO."
Reverse levels 1,2:  he said "OOF SNAEM foo."
```

Example 4:

```
Backing store:       DID YOU SAY 'he said "foo MEANS FOO"'?
Resolved levels:     1111111111122222222244433333333333221
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 that the correct appearance should be used for OPEN and CLOSE characters depending on their level: for example, OPEN PARENTHSIS appears as '(' when its resolved level is even, and as ')' when its resolved level is odd.

7. 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 support these characters must support the implicit bidiretional algorithm.

1. Supporting Explicit Formatting Codes

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

A. No bidirectional support. no right-left characters are used.

B. Implicit bidirectional support. The implicit bidirectional algorithm is supported (including RLM and LRM).

C. Full bidirectional support. Both the implicit bidirectional algorithm and the explicit directional formatting codes are supported.

2. Examples of Higher-Level Protocols

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

a. 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.

b. Override the number handling to provide for more (or less) sophisticated number parsing.

c. Supplement or override the directional overrides or embedding codes by providing information via stylesheets about the embedding level or character direction.

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

8. 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 occur when a right-to-left paragraph begins with left-to-right characters, there are nested embeddings, or when 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 of neutrals on the boundary of an embedded language. This can be addressed by correctly 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 explanation 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.
```

The other method of doing this is to type a right directional mark after the explanation mark. Since the explanation mark is now not on a directional boundary, this also produces the correct result.