Chapter 3

Conformance

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

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

3.1 Conformance Requirements

This section specifies the formal conformance requirements for processes implementing the Unicode Standard, Version 2.0. Note that this clause has been revised from the previous versions of the Unicode Standard (Versions 1.0 and 1.1). These revisions do not change the substance of the conformance requirements previously set forth but are formalized and extended to allow for the use of surrogates. Implementations that satisfied the conformance clause of the previous versions of the Unicode Standard (Versions 1.0 and 1.1) will satisfy this revised clause.

Byte Ordering

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

- This means that Unicode values can be stored in native 16-bit machine words.
- For information on transformations of Unicode text, see *Appendix A, Transformation Formats.* For information on use of `wchar_t` or other programming language types to represent Unicode values, see *Section 5.1, ANSI/ISO C* `wchar_t`.

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

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

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

- The majority of all interchange occurs with processes running on the same or a similar configuration. This makes intra-domain interchange of Unicode text in

the domain-specific byte order fully conformant and limits the role of the canonical byte order to interchange of Unicode text across domain, or where the nature of the originating domain is unknown. (For a discussion of the use of *byte order mark* to indicate byte orderings, see *Section 2.4, Special Character and Non-Character Values.)*

Invalid Code Values

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

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

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

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

Interpretation

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

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

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

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

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

- Even processes that normally do not distinguish between canonical-equivalent character sequences can have reasonable exception behavior. Some examples of this behavior include graceful fallback processing by processes unable to support correct positioning of non-spacing marks; "Show Hidden Text" modes that reveal memory representation structure; and ignoring collating behavior of combining sequences that are not part of the repertoire of a specified language.

Modification

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

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

3.2 Semantics

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

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

- Simple Properties
- Character Combination
- Canonical Decompositio
- Compatibility Decomposition
- Surrogate Property
- Bidirectional behavior, as interpreted according to the Unicode bidirectional algorithm
- Combining Jamo Behavior, as interpreted according to *Section 3.10, Combining Jamo Behavior*

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

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

- The Character Combination properties and the Canonical Ordering Behavior are not overridable by higher-level protocols.

3.3 Characters and Coded Representations

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

- When representing data, the nature of that data is generally symbolic as opposed to some other kind of data (for example, numeric, aural, or visual). Examples of such symbolic data include letters, ideographs, digits, punctuation, technical symbols, and dingbats.
- An abstract character has no concrete form.
- The abstract characters defined by the Unicode Standard are known as Unicode abstract characters.

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

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

- This term is variously referred to as a *code element*, a *code position*, a *code point*, and a *code set value* in the industry.
- The code values in the Unicode Standard are 16-bit combinations. These code values are also known as *Unicode values* for short.

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

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

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

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

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

3.4 Simple Properties

The Unicode Standard, Version 2.0 defines the normative simple character properties of *case*, *numericvalue*, *directionality*, and *mirrored*. *Chapter 4, Character Properties*, contains explicit mappings of characters to character properties. These represent the default properties for conformant processes in the absence of explicit, overriding, higher-level protocols. Additional properties that are specific to particular characters (such as the definition and use of the Right-Left Override character or zero-width spaces) are discussed in the relevant sections of this standard.

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

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

D9 *directionality property:* a property of every graphic character which determines its horizontal ordering as specified in *Section 3.11, Bidirectional Behavior.*

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

D10 *mirrored property:* the property of characters whose images are mirrored horizontally in text that is laid out from right to left (versus left to right).

- In other words, U+0028 LEFT PARENTHESIS is interpreted as an opening parenthesis; in a left-to-right context, this will appear as "(", while in a right-to-left context, this will be mirrored and appear as ")".
- This is the default behavior in Unicode text. (For more information, see the block description for the General Punctuation subsection in *Section 6.2, Symbols Area.*)

D11 *special character properties:* The behavior of most characters does not require special attention in this standard. However, certain characters exhibit special behavior, which is described in the character block descriptions. These characters are listed in *Section 3.8, Special Character Properties.*

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

3.5 Combination

D13 *base character:* a character that does not graphically combine with preceding characters.

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

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

- These are characters that are not used in isolation (unless they are being described), including such characters as accents, diacritics, Hebrew points, Arabic vowel signs, and Indic matras.
- Even though a combining character is intended to be presented in graphical combination with base character, circumstances may arise where either (1) no base character precedes the combining character, or (2) a process is unable to perform graphical combination. In both of these cases, a process may present a combining character without graphical combination; that is, it may present it as if it were a base character.
- The representative images of combining characters are depicted with a dotted circle in the code charts; when presented in graphical combination with a pre-

ceding base character, that base character is intended to appear in the position occupied by the dotted circle.

- Combining characters take on the properties of their base character (except for their combining property).

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

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

D16 *spacing mark:* a combining character that is not a non-spacing mark.

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

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

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

3.6 Decomposition

D18 *decomposable character:* a character that is equivalent to a sequence of one or more other characters, according to the decomposition mappings found in the names list of *Section 7.1, Character Names List Entries.* This may also be known as a *precomposed* character or *composite* character.

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

Compatibility Decomposition

D20 *compatibility decomposition:* the decomposition of a character which results from recursively applying *both* the compatibility mappings *and* the canonical mappings found in the names list of *Section 7.1, Character Names List Entries* until no characters can be further decomposed, and then reordering non-spacing marks according to *Section 3.9, Canonical Ordering Behavior.*

- A compatibility decomposition may remove formatting information.

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

- Compatibility characters are included in the Unicode Standard only to represent distinctions in other base standards and would not otherwise have been encoded. However, replacing a compatibility character by its decomposition may lose round-trip convertibility with a base standard.

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

Canonical Decomposition

D23 *canonical decomposition:* the decomposition of a character which results from recursively applying the canonical mappings found in the names list of *Section 7.1, Character Names List Entries* until no characters can be further decomposed, and then reordering non-spacing marks according to *Section 3.9, Canonical Ordering Behavior.*

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

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

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

3.7 Surrogates

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

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

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

- Unlike combining characters, which have independent semantics and properties, high- and low-surrogates have no interpretation when they do not appear as part of a surrogate pair, other than being a surrogate.
- Surrogate pairs are designed to allow represention of rare characters in future extensions of the Unicode Standard. There are no such currently assigned characters in this version of the standard. (For more information, see *Section 6.6, Surrogates Area.*)

D28 *Unicode scalar value:* a number N from 0 to $10FFFF_{16}$ defined by applying the following algorithm to a character sequence S (the numeric constants are in hexadecimal):

N = U	If S is a single, non-surrogate value <U>
N = (H – D800) * 400 + (L – DC00) + 10000	If S is a surrogate pair <H, L>

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

3.8 Special Character Properties

The behavior of most characters does not require special attention in this standard. However, the following characters exhibit special behavior, which is described in *Chapter 6, Character Block Descriptions*, and in *Chapter 4, Character Properties.*

- Line boundary control

```
FEFF    ZERO WIDTH NO-BREAK SPACE
2011    NON-BREAKING HYPHEN
0020    SPACE
00A0    NO-BREAK SPACE
2000    EN QUAD
2002    EN SPACE
2003    EM SPACE
2004    THREE-PER-EM SPACE
2005    FOUR-PER-EM SPACE
2006    SIX-PER-EM SPACE
2007    FIGURE SPACE
2008    PUNCTUATION SPACE
2009    THIN SPACE
200A    HAIR SPACE
200B    ZERO WIDTH SPACE
```

- Hyphenation control

```
00AD    SOFT HYPHEN
2010    HYPHEN
2011    NON-BREAKING HYPHEN
```

- Text separators

```
2028    LINE SEPARATOR
2029    PARAGRAPH SEPARATOR
0009    HORIZONTAL TAB
```

- Fraction formatting

```
2044    FRACTION SLASH
```

- Special behavior with non-spacing marks

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

- Double non-spacing marks

```
0360    COMBINING DOUBLE TILDE
0361    COMBINING DOUBLE INVERTED BREVE
```

- Joining

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

- Bidirectional ordering

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

- Alternate formatting

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

- Indic dead-character formation

094D	DEVANAGARI SIGN VIRAMA
09CD	BENGALI SIGN VIRAMA
0A4D	GURMUKHI SIGN VIRAMA
0ACD	GUJARATI SIGN VIRAMA
0B4D	ORIYA SIGN VIRAMA
0BCD	TAMIL SIGN VIRAMA
0C4D	TELUGU SIGN VIRAMA
0CCD	KANNADA SIGN VIRAMA
0D4D	MALAYALAM SIGN VIRAMA
0F84	TIBETAN SIGN HALANTA

- Code conversion fallback

FFFD	REPLACEMENT CHARACTER

- Byte order control

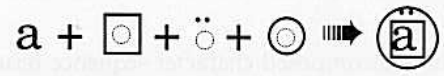
FEFF	ZERO WIDTH NO-BREAK SPACE

3.9 Canonical Ordering Behavior

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

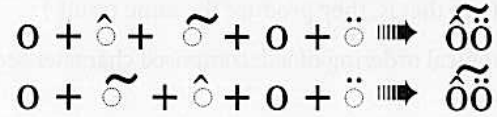
- In the Unicode Standard, all combining characters are encoded following the base characters to which they apply. The Unicode sequence U+0061 LATIN SMALL LETTER A "a" + U+0308 COMBINING DIAERESIS "¨" + U+0075 LATIN SMALL LETTER U "u" is unambiguously interpreted (and displayed) as "äu", not "aü."
-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

Figure 3-1. Enclosing Marks



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

Figure 3-2. Positioning of Double Diacritics



- Combining marks with the same combining class are generally positioned graphically outwards from the base character they modify. Some specific non-spacing marks override the default stacking behavior by being positioned side-by-side rather than stacking, or by ligaturing with an adjacent non-spacing mark. When positioned side-by-side, the order of codes is is reflected by positioning in the dominant order of the script with which they are used.
- If combining characters have different combining classes—for example, when one non-spacing mark is above a base character form and another is below—then no distinction of graphic form nor semantic will result.

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

Combining Classes

The Unicode Standard treats sequences of non-spacing marks as equivalent if they do not typographically interact. The Canonical Ordering Algorithm defines a method for determining which sequences interact and a canonical ordering of these sequences for use in equivalence comparisons.

D29 *combining class:* a numeric value given to each combining Unicode character that determines which other combining characters it typographically interacts with.

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

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

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

Collation processes may not require correct sorting outside of a given domain and may not choose to invoke the canonical ordering algorithm for excluded characters. For example, a Greek collation process may not need to sort Cyrillic letters properly; in that case, it does not have to maximally decompose and reorder Cyrillic letters and may just choose to sort them according to Unicode order.

Canonical Ordering

The canonical ordering of a decomposed character sequence results essentially by sorting each sequence of non-spacing marks according to their combining class. However, enclosing non-spacing marks are treated like base characters. Base characters never sort relative to one another, so the amount of work in the algorithm depends on the number of non-spacing marks in a row. With few occurrances of more than one non-spacing mark in a row, an implementation of this algorithm will be extremely fast. (The algorithm represents a logical description of the process. Optimized algorithms can be used in implementations as long as they are equivalent; that is, they produce the same result.)

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

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

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

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

Examples of this ordering appear in Table 3-1.

Table 3-1. Sample Combining Classes

Combining class	Abbreviation	Code	Unicode Name
0	a	0061	LATIN SMALL LETTER A
220	underdot	0323	COMBINING DOT BELOW
230	diaeresis	0308	COMBINING DIAERESIS
230	breve	0306	COMBINING BREVE
0	a-underdot	1EA0	LATIN CAPITAL LETTER A WITH DOT BELOW
0	a-diaeresis	00C4	LATIN CAPITAL LETTER A WITH DIAERESIS
0	a-breve	0102	LATIN CAPITAL LETTER A WITH BREVE

a + underdot + diaeresis ⇨ *a + underdot + diaeresis*

a + diaeresis + underdot ⇨ *a + underdot + diaeresis*

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

a + breve + diaeresis ⊗ *a + diaeresis + breve*

a + diaeresis + breve ⊗ *a + breve + diaeresis*

Thus, we get the results shown in Table 3-2 when applying the algorithm.

Table 3-2. Canonical Ordering Results

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

3.10 Combining Jamo Behavior

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

- determine the syllable boundaries in a sequence of conjoining jamo characters
- compose jamo characters into Hangul syllables
- decompose Hangul syllables into a sequence of jamo characters
- algorithmically determine the names of the Hangul syllable characters

(For more information, see *Section 6.5, Hangul Syllables Area,* and the Hangul Jamo subsection in *Section 6.1, General Scripts Area.*)

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

Syllable Boundaries

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

Within any sequence of characters, a syllable break occurs between the pairs of characters shown in Table 3-3. All other sequences of Hangul jamo are considered to be part of the same syllable.

Table 3-3. Hangul Syllable Break Rules

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

Canonical Syllables

A canonical syllable block is composed of a sequence of *choseong* followed by a sequence of *jungseong* and optionally a sequence of *jongseong* (e.g., $S = LV$ or LVT). A sequence of non-canonical syllable blocks can be transformed into a sequence of canonical syllable blocks by inserting *choseong* fillers and *jungseong* fillers.

Examples. In Table 3.4, row (1) shows syllable breaks in a canonical sequence, row (2) shows syllable breaks in a non-canonical sequence, and row (3) shows how the sequence in (2) could be transformed into canonical form by inserting fillers into each syllable.

Table 3-4. Syllable Break Examples

No.	Sequence		Sequence with syllable breaks marked
1	LVTLVLVLV_fL$_f$VL$_f$V$_f$T	→	LVT · LV · LV · LV$_f$ · L$_f$V · L$_f$V$_f$T
2	LLTVLTLTVVLL	→	LLT · V · LT · LT · VV · LL
3	LLTVLTLTVVLL	→	LLV$_f$T · L$_f$V · LV$_f$T · LV$_f$T · L$_f$VV · LLV$_f$

Hangul Syllable Composition

The following algorithm describes how to take a sequence of characters C and compose Hangul syllables. First define the following constants (the first four are hexadecimal Unicode character values and the remainder are decimal).

```
SBase  = AC00
LBase  = 1100
VBase  = 1161
TBase  = 11A7
```

```
SCount = 11172
LCount = 19
VCount = 21
TCount = 28
NCount = VCount * TCount
```

1. Process C by composing the conjoining jamo wherever possible, according to the decomposition rules in *Chapter 7, Code Charts.* (Typical interchange of conjoining jamo will be in precomposed forms. In such cases, this step not be necessary. Raw keyboard data, on the other hand, may be in decomposed form.)

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

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

3. If either of the first two characters are not in bounds (neither 0 <= LIndex < LCount nor 0 <= VIndex < VCount), then increment *i* and continue.

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

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

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

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

Example:

With the first three characters being

1111 ᄑ HANGUL CHOSEONG PHIEUPH

1171 ᅱ HANGUL JUNGSEONG WI

11B6 ᆶ HANGUL JONGSEONG RIEUL-HIEUH

We compute the following indices:

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

And replace the three characters by

```
S      = [(17 * 21) + 16] * 28 + 15 + SBase
       = D4DB
       = 퓛
```

Hangul Syllable Decomposition

The following describes the reverse mapping—how to take Hangul syllable S and derive the decomposition character sequence C.

1. Compute the index of the syllable:

   ```
   SIndex = S - SBase
   ```

2. If S is in the range (0 <= S < SCount) then compute the components as follows:

```
L      = LBase + TRUNC(SIndex / NCount)
V      = VBase + TRUNC(MOD(SIndex, NCount) / TCount)
T      = TBase + MOD(SIndex, TCount)
```

3. If T = TBase, then there is no trailing character, so replace S by the sequence <L, V>. Otherwise, there is a trailing character, so replace S by the sequence <L, V, T>.

Example:

```
L      = LBase + 17
V      = VBase + 16
T      = TBase + 15
D4DB → 1111, 1171, 11B6
```

Hangul Syllable Name

The character names for Hangul syllables are derived from the decomposition by starting with the words HANGUL SYLLABLE and adding the short name of each decomposition component in order, separated by spaces (see *Section 4.4, Jamo Short Names*). For example, for U+D4DB, derive the decomposition, as shown in the preceding example. This produces the following three-character sequence

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

The character name for U+D4DB is then generated as HANGUL SYLLABLE PWIRH

3.11 Bidirectional Behavior

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

This section describes the algorithm used to determine the directionality for bidirectional Unicode text. The algorithm extends the implicit model currently employed by a number of existing implementations and adds explicit controls for special circumstances. In most cases, there is no need to include additional information with the text to obtain correct display ordering. However, when necessary, additional information can be included in the text by means of a small set of directional formatting codes.

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

The directional formatting codes are used *only* to influence the display ordering of text. In all other respects they are ignored—they have no effect on the comparison of text, nor on word breaks, parsing, or numeric analysis. The ordering of bidirectional text depends upon

the directional properties of the text. *Section 4.3, Directionality* lists the ranges of characters that have each particular directional character type.

Directional Formatting Codes

Two types of explicit codes are used to modify the standard implicit Unicode bidirectional algorithm. In addition, there are implicit ordering codes, the *right-to-left* and *left-to-right* marks. All of these codes are limited to the current 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 as seen next.

Explicit Directional Overrides. The following codes allow the bidirectional 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 in the 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 it proceeds in two main phases. The input is a stream of text, up to a block separator (such as a paragraph separator).

- 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.
- 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-numbered levels are right-to-left, and even-numbered 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 (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.

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).

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

Notice that the neutral character (space) between THE and CAR gets the level of the surrounding characters. 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.

Combining characters always attach to preceding base character in the memory representation; this is logically *before* the bidirectional algorithm is applied. Hence, the glyph representing a combining character does not necessarily attach to the glyph, which is visually on its left in Arabic and Hebrew text. 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.

Bidirectional Character Types

For the purpose of the bidirectional algorithm, characters have the types shown in Table 3-5. (For a specification of the bidirectional character types for a given Unicode value, see *Chapter 4, Character Properties.)*

Table 3-5. Bidirectional Character Types

Type	*Category*	*Description*	*Scope*
L	Strong	Left-to-Right	Most alphabetic, syllabic, Han ideographic characters, LRM.
R	Strong	Right-to-Left	Arabic and Hebrew alphabets, punctuation specific to those scripts, RLM
EN	Weak	European Number	European digits, Eastern Arabic-Indic digits, ...
ES	Weak	European Number Separator	Figure Space, Full Stop (Period), Solidus (Slash), ...
ET	Weak	European Number Terminator	Plus Sign, Minus Sign, Degree, Currency symbols, ...
AN	Weak	Arabic Number	Arabic-Indic digits, Arabic decimal & thousands separators, ...
CS	Weak	Common Number Separator	Colon, Comma,...
B	Separator	Block Separator	Paragraph Separator, Line Separator
S	Separator	Segment Separator	Tab
WS	Neutral	Whitespace	Space, No-Break Space, General Punctuation Spaces,...
ON	Neutral	Other Neutrals	All other characters

➥ 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 subsection in *Section 6.1, General Scripts Area,* for more details on naming digits.)

Table 3-6 lists additional abbreviations used in the examples.

Table 3-6. BIDI Example Abbreviations

Symbol	Description
AL	Arabic Letter
HL	Hebrew Letter
N	Neutral or Separator (B, S, WS, ON)
sot	Start of text
eot	End of text
e	The text ordering type (L or R) that matches the embedding level direction

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 resolution process consists of seven steps: determining the base level; determining explicit embedding levels and directions; determining explicit overrides; determining embedding and override terminations; resolving weak types; resolving neutral types; 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, RLE, LRE, 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, RLE, 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*. In some contexts this is also known as the *global direction* or the *block direction*.

Explicit Levels and Directions. All explicit embedding levels are determined from the embedding and override codes. The directional level indicates both 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 reset 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 reset 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 reset 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 reset 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, and so on. If the directional override status is R, then all characters become R.

Terminating Embeddings and 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.

➥ Higher level protocols may choose to interpret PDFs that occur when there is no pushed state. For example, a presentation engine may receive blocks of pro-

cessed Unicode text divided into lines. If the complexity of the text is limited by the higher-level protocol, then PDF can be interpreted significantly.

T5. All explicit directional embeddings and overrides are completely terminated at block separators. Return to the state as of B1.

All overrides and resolution of weak types and neutrals take effect within the bounds of an embedding; that is, nothing within an embedding or override will affect the character direction of codes outside of that embedding, and vice versa. The one exception is in resolving neutrals (see N4 in the subsection "Resolving Neutral Types" in this chapter).

Now that all of the directional override controls have had their effect, apply T6.

T6. Remove implicit and explicit directional formatting codes.

Resolving Weak Types. The text is now 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.

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
L, EN           →    L, EN
HL, EN          →    HL, EN
```

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

```
EN, ES, EN      →    EN, EN, EN
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, separators and terminators change to Other Neutral:

```
AN, ET          →    AN, N
L, ES, EN       →    L, N, EN
EN, CS, AN      →    EN, N, AN
...
```

Resolving Neutral Types. 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.

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 (eot) and start-of-text (sot) 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. For the purpose of resolving neutrals,
(a) European numbers are treated as though they were the type of the previous strong letter.
(b) If there is no previous strong letter, European number are treated as though they had the base direction.
(c) Arabic numbers are treated as though they were R but do not affect the treatment of European numbers as in (a) and (b).
The following are examples:

```
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 e AN e 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 e LRO <text> PDF L L
R N [RLE <text> PDF] N L → R R RLE <text> PDF e 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".
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 since they are not surrounded on both sides (see number parsing). However, if there is an adjacent left-to-right sequence, then European numbers will adopt that direction:

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

Resolving Implicit Levels. In the final phase, the embedding level of text may be increased, based upon the resolved character type. Right-to-left text will always 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.

Table 3-7 summarizes the results of the implicit algorithm. The "(L)" indicates a preceding character type.

Table 3-7. Resolving Implicit Levels

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 algorithm 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, this 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 rule, this means that trailing white space will appear at the visual end of the line (in the base direction). Tabulation will always have a consistent 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 following four examples illustrate this:

```
Memory:              car means CAR.
Resolved levels:     00000000001110
Reverse level 1:     car means RAC.

Memory:              car MEANS CAR.
Resolved levels:     22211111111111
Reverse level 2:     rac MEANS CAR.
Reverse levels 1,2:  .RAC SNAEM car

Memory:              he said "car MEANS CAR."
Resolved levels:     00000000022211111111100
Reverse level 2:     he said "rac MEANS CAR."
Reverse levels 1,2:  he said "RAC SNAEM car."

Memory:              DID YOU SAY 'he said "car MEANS CAR"'?
Resolved levels:     11111111111122222222244433333333333211
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`

A character that possesses the mirrored property as specified by *Section 4.7, Mirrored,* should be depicted by a mirrored glyph if the resolved directionality of that character is odd. For example, U+0028 LEFT PARENTHESIS—which is interpreted in the Unicode Standard as an opening parenthesis—appears as "(" when its resolved level is even, and as the mirrored glyph ")" when its resolved level is odd.

Bidirectional Conformance

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

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, conforming systems will fall into three 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 implicit directional marks RLM and LRM are supported.
- Full bidirectionality. Both the implicit bidirectional algorithm and both the implicit and explicit directional formatting codes are supported: RLM, LRM, LRE, RLE, LRO, RLO, PDF.

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, such as 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 may require fancy text information such as 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.

Vertical Text. In the case of vertical line orientation, these formatting codes and the rest of the bidirectional algorithm is inoperative if the glyphs are ordered uniformly from top to bottom. The bidirectional algorithm applies where characters can have different ordering directions. In the case of Hebrew text, vertical lines usually 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. If embedded Latin text and numbers are rotated 90° clockwise (so that the characters are also ordered from top to bottom), then all characters in the line are also ordered from top to bottom. In all of these cases, all the characters have the same ordering direction, so the bidirectional algorithm does not apply.

The bidirectional algorithm *does* come into effect when some characters are ordered from bottom to top. For example, this happens with a mixture of Arabic and Latin glyphs when all the glyphs are rotated uniformally 90° clockwise. (The choice of whether text is to be presented horizontally or vertically, or whether text is to be rotated, is not specified by the Unicode Standard, and is left up to higher-level protocols.)

Usage

Because of the implicit character types and the heuristics for resolving neutral and numeric directional behavior, the implicit bidirectional ordering will generally produce the correct display without any further work. However, problematic cases may occur when a right-to-left paragraph begins with left-to-right characters, or 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 "MEANS CAR!", and expired.`
Display:	`he said "RAC SNAEM!", and expired.`

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

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

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