This PDF file is an excerpt from *The Unicode Standard, Version 4.0*, issued by the Unicode Consortium and published by Addison-Wesley. The material has been modified slightly for this online edition, however the PDF files have not been modified to reflect the corrections found on the Updates and Errata page (http://www.unicode.org/errata/). For information on more recent versions of the standard, see http://www.unicode.org/standard/versions/enumeratedversions.html.

Dai Kan-Wa Jiten used as the source of reference Kanji codes was written by Tetsuji Morohashi and published by Taishukan Shoten.

Cover and CD-ROM label design: Steve Mehallo, http://www.mehallo.com

Library of Congress Cataloging-in-Publication Data
The Unicode Standard, Version 4.0 : the Unicode Consortium /Joan Aliprand... [et al.].
p. cm.
Includes bibliographical references and index.
ISBN 0-321-18578-1 (alk. paper)
1. Unicode (Computer character set). I. Aliprand, Joan.

QA268.U545 2004
005.7'2—dc21

2003052158

ISBN 0-321-18578-1
Text printed on recycled paper
1 2 3 4 5 6 7 8 9 10—CRW—0706050403
First printing, August 2003

Chapter 15

Special Areas and Format Characters

This chapter describes several kinds of characters that have special properties as well as areas of the codespace that are set aside for special purposes:

- Control codes
- Layout controls
- Invisible operators
- Deprecated format characters
- Surrogates area
- Variation selectors
- Private-use characters
- Noncharacters
- Specials
- Tag characters

In addition to regular characters, the Unicode Standard contains a number of characters that are not normally rendered directly, but that influence the layout of text or otherwise affect the operation of text processes. They are called format characters.

The Unicode Standard contains code positions for the 64 control characters and the DEL character found in ISO standards and many vendor character sets. The choice of control function associated with a given character code is outside the scope of the Unicode Standard, with the exception of those control characters specified in this chapter.

Layout controls are not themselves rendered visibly, but influence the behavior of algorithms for line breaking, word breaking, glyph selection, and bidirectional ordering.

Special invisible operator characters can be used to explicitly encode some mathematical operations, such as multiplication, which are normally implied by juxtaposition. This aids in automatic interpretation of mathematical notation.

Surrogate code points are reserved and are to be used in pairs—called surrogate pairs—to access 1,048,544 supplementary characters.

Variation selectors allow the specification of standardized variants of characters. This ability is particularly useful where the majority of implementations would treat the two variants as two forms of the same character, but where some implementations need to differentiate between the two. By using a variation selector, such differentiation can be made explicit.

Private-use characters are reserved for private use. Their meaning is defined by private agreement.

Noncharacters are code points that are permanently reserved and will never have characters assigned to them.

The Specials block contains characters that are neither graphic characters nor traditional controls.

Tag characters support a general scheme for the internal tagging of text streams in the absence of other mechanisms, such as markup languages. They are reserved for use with specific plain text-based protocols that specify their usage. Their use in ordinary text is strongly discouraged.

15.1 Control Codes

There are 65 code points set aside in the Unicode Standard for compatibility with the C0 and C1 control codes defined in the ISO/IEC 2022 framework. The ranges of these code points are U+0000..U+001F, U+007F, and U+0080..U+009F, which correspond to the 8-bit controls 00_{16} to $1F_{16}$ (C0 controls), $7F_{16}$ (*delete*), and 80_{16} to $9F_{16}$ (C1 controls), respectively. For example, the 8-bit legacy control code *character tabulation* (or *tab*) is the byte value 09_{16}; the Unicode Standard encodes the corresponding control code at U+0009.

The Unicode Standard provides for the intact interchange of these code points, neither adding to nor subtracting from their semantics. The semantics of the control codes are generally determined by the application with which they are used. However, in the absence of specific application uses, they may be interpreted according to the control function semantics specified in ISO/IEC 6429.

In general, the use of control codes constitutes a higher-level protocol and is beyond the scope of the Unicode Standard. For example, the use of ISO/IEC 6429:1992 control sequences for controlling bidirectional formatting would be a legitimate higher-level protocol layered on top of the plain text of the Unicode Standard. Higher-level protocols are not specified by the Unicode Standard; their existence cannot be assumed without a separate agreement between the parties interchanging such data.

Representing Control Sequences

There is a simple, one-to-one mapping between 7-bit (and 8-bit) control codes and the Unicode control codes: every 7-bit (or 8-bit) control code is numerically equal to its corresponding Unicode code point. For example, if the ASCII *line feed* control code ($0A_{16}$) is to be used for line break control, then the text "WX<LF>YZ" would be transmitted in Unicode plain text as the following coded character sequence: <0057, 0058, 000A, 0059, 005A>.

Control sequences that are part of Unicode text must be represented in terms of the Unicode encoding forms. For example, suppose that an application allows embedded font information to be transmitted by means of markup using plain text and control codes. A font tag specified as "^ATimes^B", where ^A refers to the C0 control code 01_{16} and ^B refers to the C0 control code 02_{16}, would then be expressed by the following coded character sequence: <0001, 0054, 0069, 006D, 0065, 0073, 0002>. The representation of the control codes in the three Unicode encoding forms simply follows the rules for any other code points in the standard:

UTF-8: <01 54 69 6D 65 73 02>

UTF-16: <0001 0054 0069 006D 0065 0073 0002>

UTF-32: <00000001 00000054 00000069 0000006D
00000065 00000073 00000002>

Escape Sequences. Escape sequences are a particular type of protocol that consists of the use of some set of ASCII characters introduced by the *escape* control code, $1B_{16}$, to convey extra-textual information. When converting escape sequences into and out of Unicode text, they should be converted on a character-by-character basis. For instance, "ESC-A" <1B 41> would be converted into the Unicode coded character sequence <001B, 0041>. Interpretation of U+0041 as part of the escape sequence, rather than as *latin capital letter a* is the responsibility of the higher-level protocol that makes use of such escape sequences. This approach allows for low-level conversion processes to conformantly convert escape sequences into and out of the Unicode Standard without needing to actually recognize the escape sequences as such.

If a process uses escape sequences or other configurations of control code sequences to embed additional information about text (such as formatting attributes or structure), then such sequences constitute a higher-level protocol outside the scope of the Unicode Standard.

Specification of Control Code Semantics

There are several control codes commonly used in plain text, particularly those involved in line and paragraph formatting. The use of these control codes is widespread and important to interoperability. Therefore, the Unicode Standard specifies semantics for their use with the rest of the encoded characters in the standard. *Table 15-1* lists those control codes.

Table 15-1. Control Codes Specified in the Unicode Standard

Code Point	Abbreviation	ISO/IEC 6429 Name
U+0009	HT	character tabulation (tab)
U+000A	LF	line feed
U+000B	VT	line tabulation (vertical tab)
U+000C	FF	form feed
U+000D	CR	carriage return
U+001C	FS	information separator four
U+001D	GS	information separator three
U+001E	RS	information separator two
U+001F	US	information separator one
U+0085	NEL	next line

Most of the control codes in *Table 15-1* have the White_Space property. They have the directional property values of S or B, rather than the default of ON used for other control codes. (See Unicode Standard Annex #9, "The Bidirectional Algorithm.") In addition, the separator semantics of the control codes U+001C..U+001F are recognized in the bidirectional algorithm. U+0009..U+000D also have line breaking property values that differ from the default CM value for other control codes. (See Unicode Standard Annex #14, "Line Breaking Properties.")

U+0000 *null* may be used as a Unicode string terminator, as in the C language. Such usage is outside the scope of the Unicode Standard, which does not require any particular formal language representation of a string, nor any particular usage of null.

Newline Function. In particular, one or more of the control codes U+000A *line feed*, U+000D *carriage return*, or the Unicode equivalent of the EBCDIC *next line* encode a *newline function*. A newline function can act like a *line separator* or a *paragraph separator*, depending on the application. See *Section 15.2, Layout Controls*, for information on how to interpret a line or paragraph separator. The exact encoding of a newline function depends on the application domain. For information on how to identify a newline function, see *Section 5.8, Newline Guidelines*.

15.2 Layout Controls

The effect of layout controls is specific to particular text processes. As much as possible, layout controls are transparent to those text processes for which they were not intended. In other words, their effects are mutually orthogonal.

Line and Word Breaking

The following gives a brief summary of the intended behavior of certain layout controls. For a full description of line and word breaking layout controls, see Unicode Standard Annex #14, "Line Breaking Properties."

No-Break Space. U+00A0 NO-BREAK SPACE has the same width as U+0020 SPACE, but the NO-BREAK SPACE indicates that, under normal circumstances, no line breaks are permitted between it and surrounding characters, unless the preceding or following character is a line or paragraph separator. U+00A0 NO-BREAK SPACE behaves like the following coded character sequence: U+FEFF ZERO WIDTH NO-BREAK SPACE + U+0020 SPACE + U+FEFF ZERO WIDTH NO-BREAK SPACE. For a complete list of space characters in the Unicode Standard, see *Table 6-2*.

Word Joiner. U+2060 WORD JOINER behaves like U+00A0 NO-BREAK SPACE in that it indicates the absence of word boundaries; however, the *word joiner* has no width. The function of the character is to indicate that line breaks are not allowed between the adjoining characters, except next to hard line breaks. For example, the word joiner can be inserted after the fourth character in the text "base+delta" to indicate that there should be no line break between the "e" and the "+". The *word joiner* can be used to prevent line breaking with other characters that do not have nonbreaking variants, such as U+2009 THIN SPACE or U+2015 HORIZONTAL BAR, by bracketing the character.

The word joiner must not be confused with the *zero width joiner* or the *combining grapheme joiner*, which have very different functions. In particular, inserting a word joiner between two characters has no effect on their ligating and cursive joining behavior. The word joiner should be ignored in contexts other than word or line breaking.

Zero Width No-Break Space. In addition to its primary meaning of *byte order mark* (see "Byte Order Mark" in *Section 15.9, Specials*), the code point U+FEFF possesses the semantics of ZERO WIDTH NO-BREAK SPACE, which matches that of *word joiner*. Until Unicode 3.1.1, U+FEFF was the only code point with word joining semantics, but because it is more commonly used as *byte order mark,* the use of U+2060 WORD JOINER to indicate word joining is strongly preferred for any new text. Implementations should continue to support the word joining semantics of U+FEFF for backward compatibility.

Zero Width Space. The U+200B ZERO WIDTH SPACE indicates a word boundary, except that it has no width. Zero-width space characters are intended to be used in languages that have no visible word spacing to represent word breaks, such as Thai, Khmer, or Japanese. When text is justified, ZWSP has no effect on letter spacing—for example, in English or Japanese usage.

There may be circumstances with other scripts, such as Thai, where extra space is applied around ZWSP as a result of justification, as shown in *Figure 15-1*. This approach is unlike the use of fixed-width space characters, such as U+2002 EN SPACE, that have specified width and should not be automatically expanded during justification (see *Section 6.2, General Punctuation*).

Figure 15-1. Letter Spacing

Type	Justification Examples	Explanation
Memory	the ISP®[ZWSP]Charts	The [ZWSP] is inserted to allow linebreak after ®
Display 1	the ISP®Charts	Without letter spacing
Display 2	t h e I S P ® C h a r t s	Increased letter spacing
Display 3	t h e I S P ® C h a r t s	"Thai-style" letter spacing
Display 4	t h e I S P ®C h a r t s	[ZWSP] incorrectly inhibiting letter spacing (after ®)

In some languages such as German or Russian, increased letter spacing is used to indicate emphasis. Implementers should be aware of this issue.

Zero-Width Spaces and Joiner Characters. The zero-width spaces are not to be confused with zero-width joiner characters. U+200C ZERO WIDTH NON-JOINER and U+200D ZERO WIDTH JOINER have no effect on word boundaries, and ZERO WIDTH NO-BREAK SPACE and ZERO WIDTH SPACE have no effect on joining or linking behavior. In other words, the zero-width joiner characters should be ignored when determining word boundaries; ZERO WIDTH SPACE should be ignored when determining cursive joining behavior. See "Cursive Connection" later in this section.

Hyphenation. U+00AD SOFT HYPHEN (SHY) indicates an intraword break point, where a line break is preferred if a word must be hyphenated or otherwise broken across lines. Such break points are generally determined by an automatic hyphenator. The use of SHY is generally limited to situations where users need to override the behavior of such a hyphenator. The visible rendering of a line break at an intraword break point, whether automatically determined or indicated by a SHY, depends on the surrounding characters, the language, and, at times, the meaning of the word. The precise rules are outside the scope of this standard, but see Unicode Standard Annex #14, "Line Breaking Properties," for additional information. A common default rendering is to insert a hyphen before the line break, but this is incorrect in many situations.

Contrast this usage with U+2027 HYPHENATION POINT, which is used for a visible indication of the place of hyphenation in dictionaries. For a complete list of dash characters, including all the hyphens, in the Unicode Standard, see *Table 6-3*.

There are two nonbreaking hyphen characters in the Unicode Standard: U+2011 NON-BREAKING HYPHEN and U+0F0C TIBETAN MARK DELIMITER TSHEG BSTAR. See *Section 9.11, Tibetan*, for more discussion of the Tibetan-specific line breaking behavior.

Line and Paragraph Separator. The Unicode Standard provides two unambiguous characters, U+2028 LINE SEPARATOR and U+2029 PARAGRAPH SEPARATOR, to separate lines and paragraphs. They are considered the default form of denoting line and paragraph boundaries in Unicode plain text. A new line is begun after each LINE SEPARATOR. A new paragraph is begun after each PARAGRAPH SEPARATOR. As these characters are separator codes, it is not necessary either to start the first line or paragraph or to end the last line or paragraph with them. Doing so would indicate that there was an empty paragraph or line following. The PARAGRAPH SEPARATOR can be inserted between paragraphs of text. Its use allows the creation of plain text files, which can be laid out on a different line width at the receiving end. The LINE SEPARATOR can be used to indicate an unconditional end of line.

A paragraph separator indicates where a new paragraph should start. Any interparagraph formatting would be applied. This formatting could cause, for example, the line to be broken, any interparagraph line spacing to be applied, and the first line to be indented. A *line separator* indicates that a line break should occur at this point; although the text continues on the next line, it does not start a new paragraph—no interparagraph line spacing or paragraphic indentation is applied. For more information on line separators, see *Section 5.8, Newline Guidelines*.

Cursive Connection and Ligatures

In some fonts for some scripts, consecutive characters in a text stream may be rendered via adjacent glyphs that cursively join to each other, so as to emulate connected handwriting. For example, cursive joining is implemented in nearly all fonts for the Arabic scripts and in a few handwriting-like fonts for the Latin script.

Cursive rendering is implemented by joining glyphs in the font, as well as by using a process that selects the particular joining glyph to represent each individual character occurrence, based on the joining nature of its neighboring characters. This glyph selection is implemented in the rendering engine, typically using information in the font.

In many cases there is an even closer binding, where a sequence of characters is represented by a single glyph, called a ligature. Ligatures can occur in both cursive and noncursive fonts. Where ligatures are available, it is the task of the rendering system to select a ligature to create the most appropriate line layout. However, the rendering system cannot define the locations where ligatures are possible because there are many languages in which ligature formation requires more information. For example, in some languages ligatures are never formed across syllable boundaries.

On occasion an author may wish to override the normal automatic selection of connecting glyphs or ligatures. Typically, this choice is made to achieve one of the following effects:

- Cause nondefault joining appearance (for example, as is sometimes required in writing Persian using the Arabic script)
- Exhibit the joining-variant glyphs themselves in isolation
- Request a ligature to be formed where it normally would not be
- Request a ligature not to be formed where it normally would be

The Unicode Standard provides a way to influence joining and ligature glyph selection, by means of the two characters U+200C ZERO WIDTH NON-JOINER and U+200D ZERO WIDTH JOINER. The zero-width joiner and non-joiner request a rendering system to have more or less of a connection between characters than they would otherwise have. Such a connection may be a simple cursive link, or it may include control of ligatures.

The cursive joiner and non-joiner characters are designed for use in plain text; they should not be used where higher-level ligation and cursive control is available. (See Unicode Technical Report #20, "Unicode in XML and Other Markup Languages," for more information.) Moreover, they are essentially requests for the rendering system to take into account when laying out the text; while a rendering system should take them into account, it is perfectly acceptable for the system to disregard these requests.

The ZWJ and ZWNJ are designed for marking the unusual cases where ligatures or cursive connections are required or prohibited. These characters are not to be used in all cases where ligatures or cursive connections are desired; instead, they are only for overriding the normal behavior of the text.

Joiner. U+200D zero width joiner is intended to produce a more connected rendering of adjacent characters than would otherwise be the case, if possible. In particular:

- If the two characters could form a ligature but do not normally, ZWJ requests that the ligature be used.
- Otherwise, if either of the characters could cursively connect but do not normally, ZWJ requests that each of the characters take a cursive-connection form where possible.

In a sequence like <X, ZWJ, Y>, where a cursive form exists for X but not for Y, the presence of ZWJ requests a cursive form for X. Otherwise, where neither a ligature nor a cursive connection is available, the ZWJ has no effect. In other words, given three broad categories below, ZWJ requests that glyphs in the highest available category (for the given font) be used:

1. Unconnected
2. Cursively connected
3. Ligated

Non-Joiner. U+200C zero width non-joiner is intended to break both cursive connections and ligatures in rendering.

ZWNJ requests that glyphs in the lowest available category (for the given font) be used.

For those unusual circumstances where someone wants to forbid ligatures in a sequence XY, but promote cursive connection, the sequence <X, ZWJ, ZWNJ, ZWJ, Y> can be used. The ZWNJ breaks ligatures, while the two adjacent joiners cause the X and Y to take adjacent cursive forms (where they exist). Similarly, if someone wanted to have X take a cursive form but Y be isolated, then the sequence <X, ZWJ, ZWNJ, Y> could be used (as in previous versions of the Unicode Standard). Examples are shown in *Figure 15-2*.

Cursive Connection. For cursive connection, the joiner and non-joiner characters typically do not modify the contextual selection process itself, but instead change the context of a particular character occurrence. By providing a non-joining adjacent character where the adjacent character otherwise would be joining, or vice versa, they deceive the rendering process into selecting a different joining glyph. This process can be used in two ways:

1. Prevent joining appearance. For example,

ص	U+0635 arabic letter sad
ZWNJ	U+200C zero width non-joiner
ل	U+0644 arabic letter lam

would be rendered as ص‌ل (that is, the normal cursive joining of the interior *sad* and *lam* is overridden). Without the zero width non-joiner, it would be rendered as صل

2. Exhibit joining glyphs in isolation. For example,

ZWJ	U+200D zero width joiner
غ	U+063A arabic letter ghain
ZWJ	U+200D zero width joiner

would be rendered as ـغـ (that is, the medial glyph form of the *ghain* appears in isolation). Without the ZERO WIDTH JOINER before and after, it would be rendered as غ.

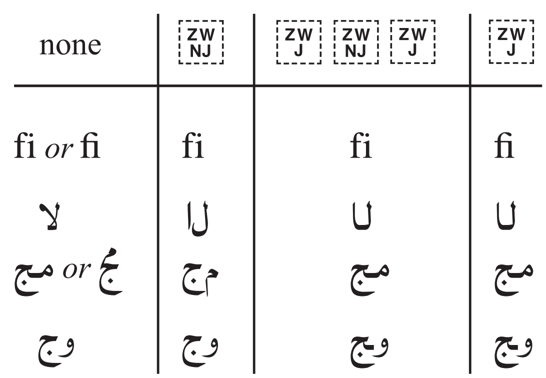
The preceding examples are adapted from the Iranian national coded character set standard, ISIRI 3342, which defines these characters as “pseudo space” and “pseudo connection,” respectively.

The ZERO WIDTH JOINER and ZERO WIDTH NON-JOINER also have specific interpretations in certain scripts as specified in this standard. For example, in Indic scripts the ZERO WIDTH JOINER provides an invisible neighbor to which a dead consonant may join to induce a half-consonant form (see *Section 9.1, Devanagari*).

ZERO WIDTH NON-JOINER and ZERO WIDTH JOINER are format control characters. Like other such characters, they should be ignored by processes that analyze text content. For example, a spelling-checker or find/replace operation should filter them out. (See *Section 2.11, Special Characters and Noncharacters*, for a general discussion of format control characters.)

Examples. *Figure 15-2* provide samples of desired renderings when the joiner or non-joiner is inserted between two characters. In the Arabic examples, the characters on the left side are in visual order already, but have not yet been shaped. The examples presume that all of the glyphs are available in the font. If, for example, the ligatures are not available, the display would fall back to the unligated forms.

Figure 15-2. Sample Display Actions



For backward compatibility, between Arabic characters a ZWJ acts just like the sequence <ZWJ, ZWNJ, ZWJ>, preventing a ligature from forming instead of requesting the use of a ligature that would not normally be used. As a result, there is no plain text mechanism for requesting the use of a ligature in Arabic text.

Implementation Notes. For modern font technologies, such as OpenType or AAT, font vendors should add ZWJ to their ligature mapping tables as appropriate. Thus, where a font had a mapping from “f” + “i” to ﬁ, the font designer should add the additional mapping from “f” + ZWJ + “i” to ﬁ. On the other hand, ZWNJ will normally have the desired effect naturally for most fonts without any change, as it simply obstructs the normal ligature/cursive connection behavior. As with all other alternate format characters, fonts should use an invisible zero-width glyph for representation of both ZWJ and ZWNJ.

Combining Grapheme Joiner

U+034F COMBINING GRAPHEME JOINER is used to indicate that adjacent characters are to be treated as a unit for the purposes of language-sensitive collation and searching. In language-sensitive collation and searching, the combining grapheme joiner should be ignored unless it specifically occurs within a tailored collation element mapping. Thus it is given a completely ignorable collation element in the default collation table, like NULL (see Unicode Technical Standard #10, "Unicode Collation Algorithm," and also ISO/IEC 14651). However, it can be entered into the tailoring rules for any given language, using the tailoring capabilities of the collation standards.

For rendering, the combining grapheme joiner is invisible. However, some older implementations may treat a sequence of grapheme clusters linked by combining grapheme joiners as a single unit for the application of enclosing combining marks. For more information on grapheme clusters, see Unicode Technical Report #29, "Text Boundaries." For more information on enclosing combining marks, see *Section 3.11, Canonical Ordering Behavior*.

The combining grapheme joiner must not be confused with the *zero width joiner* or the *word joiner*, which have very different functions. In particular, inserting a combining grapheme joiner between two characters should have no effect on their ligation or cursive joining behavior. Where the prevention of line breaking is the desired effect, the word joiner should be used. For more information on the behavior of these characters in line breaking, see Unicode Standard Annex #14, "Line Breaking Properties."

Bidirectional Ordering Controls

Bidirectional ordering controls are used in the bidirectional algorithm, described in Unicode Standard Annex #9, "The Bidirectional Algorithm." Systems that handle right-to-left scripts such as Arabic, Syriac, and Hebrew, for example, should be sensitive to these format control characters. The bidirectional ordering controls are shown in *Table 15-2*.

Table 15-2. Bidirectional Ordering Controls

Code	Name	Abbreviation
U+200E	LEFT-TO-RIGHT MARK	LRM
U+200F	RIGHT-TO-LEFT MARK	RLM
U+202A	LEFT-TO-RIGHT EMBEDDING	LRE
U+202B	RIGHT-TO-LEFT EMBEDDING	RLE
U+202C	POP DIRECTIONAL FORMATTING	PDF
U+202D	LEFT-TO-RIGHT OVERRIDE	LRO
U+202E	RIGHT-TO-LEFT OVERRIDE	RLO

As with other format control characters, bidirectional ordering controls affect the layout of the text in which they are contained, but should be ignored for other text processes, such as sorting or searching. However, text processes that modify text content must maintain these characters correctly, because matching pairs of bidirectional ordering controls must be coordinated, so as not to disrupt the layout and interpretation of bidirectional text. Each instance of a LRE, RLE, LRO, or RLO is normally paired with a corresponding PDF.

U+200E LEFT-TO-RIGHT MARK and U+200F RIGHT-TO-LEFT MARK have the semantics of an invisible character of zero width, except that these characters have strong directionality. They are intended to be used to resolve cases of ambiguous directionality in the context of bidirectional texts and are not paired. Unlike U+200B ZERO WIDTH SPACE, these characters carry no word breaking semantics. (See Unicode Standard Annex #9, "The Bidirectional Algorithm," for more information.)

15.3 Invisible Operators

The General Punctuation block contains several special format control characters known as *invisible operators*, for use in machine interpretation of mathematical expressions. Those characters, encoded in the range U+2061..U+2063, are discussed in this section.

In mathematics some operators and punctuation are often implied but not displayed. U+2063 INVISIBLE SEPARATOR (also known as *invisible comma*) is intended for use in index expressions and other mathematical notation where two adjacent variables form a list and are not implicitly multiplied. In mathematical notation commas are not always explicitly present, but need to be indicated for symbolic calculation software to help it disambiguate a sequence from a multiplication. For example, the double $_{ij}$ subscript in the variable a_{ij} means $a_{i,j}$—that is, the *i* and *j* are separate indices and not a single variable with the name *ij* or even the product of *i* and *j*. To represent the implied list separation in the subscript $_{ij}$, one can insert a nondisplaying *invisible separator* between the *i* and the *j*. In addition, use of the invisible comma would hint to a math layout program to typeset a small space between the variables.

Similarly, an expression like mc^2 implies that the mass *m* multiplies the square of the speed *c*. To represent the implied multiplication in mc^2, one inserts a nondisplaying U+2062 INVISIBLE TIMES between the *m* and the *c*. A related case is the use of U+2061 FUNCTION APPLICATION for an implied function dependence, as in $f(x + y)$. To indicate that this is the function *f* of the quantity $x + y$ and not the expression $fx + fy$, one can insert the nondisplaying *function application symbol* between the *f* and the left parenthesis.

Another example can be seen in the expression $f^{ij}(\cos(ab))$, which has the same meaning as $f^{ij}(\cos(a\times b))$, where × represents *multiplication*, not the *cross product*. Note that the spacing between characters may also depend on whether the adjacent variables are part of a list or are to be concatenated (that is, multiplied).

Use of invisible operators is optional and is intended for interchange with math-aware programs.

A more complete discussion of mathematical notation can be found in Unicode Technical Report #25, "Unicode Support for Mathematics."

15.4 Deprecated Format Characters

Deprecated Format Characters: U+206A–U+206F

Three pairs of deprecated format characters are encoded in this block:

- Symmetric swapping format characters used to control the glyphs that depict characters such as "(" (The default state is *activated.*)
- Character shaping selectors used to control the shaping behavior of the Arabic compatibility characters (The default state is *inhibited.*)
- Numeric shape selectors used to override the normal shapes of the Western digits (The default state is *nominal.*)

The use of these character shaping selectors and codes for digit shapes is *strongly* discouraged in the Unicode Standard. Instead, the appropriate character codes should be used with the default state. For example, if contextual forms for Arabic characters are desired, then the nominal characters should be used, and not the presentation forms with the shaping selectors. Similarly, if the Arabic digit forms are desired, then the explicit characters should be used, such as U+0660 ARABIC-INDIC DIGIT ZERO.

Symmetric Swapping. The symmetric swapping format characters are used in conjunction with the class of left- and right-handed pairs of characters (symmetric characters), such as parentheses. The characters thus affected are listed in *Section 4.7, Bidi Mirrored—Normative*. They indicate whether the interpretation of the term LEFT or RIGHT in the character names should be interpreted as meaning *opening* or *closing*, respectively. They do not nest. The default state of symmetric swapping may be set by a higher-level protocol or standard, such as ISO 6429. In the absence of such a protocol, the default state is *activated*.

From the point of encountering U+206A INHIBIT SYMMETRIC SWAPPING format character up to a subsequent U+206B ACTIVATE SYMMETRIC SWAPPING (if any), the symmetric characters will be interpreted and rendered as left and right.

From the point of encountering U+206B ACTIVATE SYMMETRIC SWAPPING format character up to a subsequent U+206A INHIBIT SYMMETRIC SWAPPING (if any), the symmetric characters will be interpreted and rendered as opening and closing. This state (*activated*) is the default state in the absence of any symmetric swapping code or a higher-level protocol.

Character Shaping Selectors. The character shaping selector format characters are used in conjunction with Arabic presentation forms. During the presentation process, certain letterforms may be joined together in cursive connection or ligatures. The shaping selector codes indicate that the character shape determination (glyph selection) process used to achieve this presentation effect is to be either activated or inhibited. The shaping selector codes do not nest.

From the point of encountering a U+206C INHIBIT ARABIC FORM SHAPING format character up to a subsequent U+206D ACTIVATE ARABIC FORM SHAPING (if any), the character shaping determination process should be inhibited. If the backing store contains Arabic presentation forms (for example, U+FE80..U+FEFC), then these forms should be presented without shape modification. This state (*inhibited*) is the default state in the absence of any character shaping selector or a higher-level protocol.

From the point of encountering a U+206D ACTIVATE ARABIC FORM SHAPING format character up to a subsequent U+206C INHIBIT ARABIC FORM SHAPING (if any), any Arabic presentation forms that appear in the backing store should be presented with shape modification by means of the character shaping (glyph selection) process.

The shaping selectors have no effect on nominal Arabic characters (U+0660..U+06FF), which are always subject to character shaping (glyph selection).

Numeric Shape Selectors. The numeric shape selector format characters allow the selection of the shapes in which the digits U+0030..U+0039 are to be rendered. These format characters do not nest.

From the point of encountering a U+206E NATIONAL DIGIT SHAPES format character up to a subsequent U+206F NOMINAL DIGIT SHAPES (if any), the European digits (U+0030.. U+0039) should be depicted using the appropriate national digit shapes as specified by means of appropriate agreements. For example, they could be displayed with shapes such as the ARABIC-INDIC DIGITS (U+0660..U+0669). The actual character shapes (glyphs) used to display national digit shapes are not specified by the Unicode Standard.

From the point of encountering a U+206F NOMINAL DIGIT SHAPES format character up to a subsequent U+206E NATIONAL DIGIT SHAPES (if any), the European digits (U+0030.. U+0039) should be depicted using glyphs that represent the nominal digit shapes shown in the code tables for these digits. This state (*nominal*) is the default state in the absence of any numeric shape selector or a higher-level protocol.

15.5 Surrogates Area

Surrogates Area: U+D800–U+DFFF

When using UTF-16 to represent supplementary characters, pairs of 16-bit code units are used for each character. These units are called *surrogates*. To distinguish them from ordinary characters, they are allocated in a separate area. The Surrogates Area consists of 1,024 low-half surrogate code points and 1,024 high-half surrogate code points. For the formal definition of a *surrogate pair* and the role of surrogate pairs in the Unicode Conformance Clause, see *Section 3.8, Surrogates,* and *Section 5.4, Handling Surrogate Pairs in UTF-16*.

The use of surrogate pairs in the Unicode Standard is formally equivalent to the Universal Transformation Format-16 (UTF-16) defined in ISO 10646. For more information, see *Appendix C, Relationship to ISO/IEC 10646*. For a complete statement of UTF-16, see *Section 3.9, Unicode Encoding Forms*.

High-Surrogate. The high-surrogate code points are assigned to the range U+D800..U+DBFF. The high-surrogate code point is always the first element of a surrogate pair.

Low-Surrogate. The low-surrogate code points are assigned to the range U+DC00..U+DFFF. The low-surrogate code point is always the second element of a surrogate pair.

Private-Use High-Surrogates. The high-surrogate code points from U+DB80..U+DBFF are private-use high-surrogate code points (a total of 128 code points). Characters represented by means of a surrogate pair, where the high-surrogate code point is a private-use high-surrogate, are private-use characters from the supplementary private use areas. For more information on private-use characters see *Section 15.7, Private-Use Characters*.

The code tables do not have charts or name list entries for the range D800..DFFF because individual, unpaired surrogates merely have code points.

15.6 Variation Selectors

Characters in the Unicode Standard can be represented by a wide variety of glyphs, as discussed in *Chapter 2, General Structure*. Occasionally the need arises in text processing to restrict or change the set of glyphs that are to be used to represent a character. Normally such changes are indicated by choice of font or style in rich text documents. In special circumstances, such a variation from the normal range of appearance needs to be expressed side-by-side in the same document in plain text contexts, where it is impossible or inconvenient to exchange formatted text. For example, in languages employing the Mongolian script, sometimes a specific variant range of glyphs is needed for a specific textual purpose for which the range of "generic" glyphs is considered inappropriate. The variation selectors are used when characters have essentially the same semantic.

Variation selectors provide a mechanism for specifying a restriction on the set of glyphs that are used to represent a particular character. They also provide a mechanism for specifying variants, such as for CJK ideographs and Mongolian letters, that have essentially the same semantic but substantially different ranges of glyphs. A variation sequence, which always consists of a base character followed by the variation selector, may be specified as part of the Unicode Standard. That sequence is referred to as a *variant* of the base character. The variation selector affects *only* the appearance of the base character, and only in the variation sequences defined in this Standard. The variation selector is *not* used as a general code extension mechanism.

> *Only the variation sequences specifically defined in the file StandardizedVariants.txt in the Unicode Character Database are sanctioned for standard use; in all other cases the variation selector cannot change the visual appearance of the preceding base character from what it would have had in the absence of the variation selector.*

The base character in a variation sequence is never a combining character or a decomposable character. The variation selectors themselves are combining marks of combining class 0 and are default ignorable characters. Thus, if the variation sequence is not supported, the variation selector should be invisible and ignored. As with all default ignorable characters, this does not preclude modes or environments where the variation selectors should be given visible appearance. For example, a "Show Hidden" mode could reveal the presence of such characters with specialized glyphs, or a particular environment could use or require a visual indication of a base character (such as a wavy underline) to show that it is part of a standardized variation sequence that cannot be supported by the current font.

The standardization or support of a particular variation sequence does *not* limit the set of glyphs that can be used to represent the base character alone. If a user *requires* a visual distinction between a character and a particular variant of that character, then fonts must be used to make that distinction. The existence of a variation sequence does not preclude the later encoding of a new character with a distinct semantic and a similar or overlapping range of glyphs.

For the behavior of older implementations of Mongolian using variation selectors, see the discussion of Mongolian free variation selectors in *Section 12.2, Mongolian*.

15.7 Private-Use Characters

Private-use characters are assigned Unicode code points whose interpretation is not specified by this standard and whose use may be determined by private agreement among cooperating users. These characters are designated for private use and do not have defined, interpretable semantics except by private agreement.

No charts are provided for private-use characters, as any such characters are, by their very nature, only defined outside the context of this standard.

Three distinct blocks of private-use characters are provided in the Unicode Standard: the primary Private Use Area (PUA) in the BMP, and two supplementary Private Use Areas in the supplemental planes.

All code points in the blocks of private-use characters in the Unicode Standard are permanently designated for private use—no assignment to a particular, standard set of characters will ever be endorsed or documented by the Unicode Consortium for any of these code points.

Any prior use of a character as a private-use character has no direct bearing on any eventual encoding decisions regarding whether and how to encode that character. Standardization of characters must always follow the normal process for encoding of new characters or scripts. (See *Section 1.6, Submitting New Characters*.)

Properties. The Unicode Character Database provides default character properties, which implementations can use for the processing of private-use characters. In addition, users of private-use characters may exchange external data that allows them also to exchange private-use characters in a semantically consistent way between implementations. The Unicode Standard provides no predefined format for such an exchange.

Normalization. The canonical and compatibility decompositions of any private-use character (for example, U+E000) is the character itself. This is normatively defined by the Unicode Standard and cannot be changed by private agreement. The treatment of all private-use characters for normalization forms NFC, NFD, NFKD, and NFKC is also normatively defined by the Unicode Standard on the basis of these decompositions. (See Unicode Standard Annex #15, "Unicode Normalization Forms.") No private agreement may change these forms, for example, by changing the standard canonical or compatibility decompositions for private-use characters.

This does not preclude private agreements on other transformations. Thus one could define a transformation "MyCompanyComposition" that was identical to NFC except that it mapped U+E000 to "a". The forms NFC, NFD, NFKD, and NFKC themselves, however, cannot be changed by such agreements.

Private Use Area: U+E000–U+F8FF

The primary Private Use Area consists of code points in the range U+E000 to U+F8FF, for a total of 6,400 private-use characters.

Encoding Structure. By convention, the primary Private Use Area is divided into a corporate use subarea for platform writers, starting at U+F8FF and extending downward in values, and an end user subarea, starting at U+E000 and extending upward.

By following this convention, the likelihood of collision between private-use characters defined by platform writers with private-use characters defined by end users can be reduced. However, it should be noted that this is only a convention, not a normative specification. In principle, any user can define any interpretation of any private-use character.

Corporate Use Subarea. Systems vendors and/or software developers may need to reserve some private-use characters for internal use by their software. The corporate use subarea is the preferred area for such reservations. Assignments of character semantics in this subarea may be completely internal, hidden from end users, and used only for vendor-specific application support, or they may be published as vendor-specific character assignments available to applications and end users. An example of the former case would be the assignment of a character code to a system support operation such as <MOVE> or <COPY>; an example of the latter case would be the assignment of a character code to a vendor-specific logo character such as Apple's *apple* character.

Note, however, that in many cases systems vendors may need to support full end-user definability for all private-use characters, for such purposes as *gaiji* support or for transient cross-mapping tables. The use of noncharacters (See *Section 15.8, Noncharacters*, and definition D7b in *Section 3.4, Characters and Encoding*) is the preferred way to make use of *non-interchangeable* internal system sentinels of various sorts.

End User Subarea. The end user subarea is intended for private-use character definitions by end users or for scratch allocations of character space by end-user applications.

Allocation of Subareas. Vendors may choose to reserve ranges of private-use characters in the corporate use subarea and make some defined portion of the end user subarea available for completely free end-user definition. The convention of separating the two subareas is merely a suggestion for the convenience of system vendors and software developers. No firm dividing line between the two subareas is defined in this standard, as different users may have different requirements. No provision is made in the Unicode Standard for avoiding a "stack-heap collision" between the two subareas; in other words, there is no guarantee that end users will not define a private-use character at a code point that overlaps and conflicts with a particular corporate private-use definition at the same code point. Avoiding such overlaps in definition is up to implementations and users.

Supplementary Private Use Areas

Encoding Structure. The entire Plane 15, with the exception of the noncharacters U+FFFFE and U+FFFFF, is defined to be the Supplementary Private Use Area-A. The entire Plane 16, with the exception of the noncharacters U+10FFFE and U+10FFFF, is defined to be the Supplementary Private Use Area-B. Together these areas make an additional 131,068 code points available for private use.

The supplementary PUAs provide additional undifferentiated space for private-use characters for implementations for which the 6,400 private-use characters in the primary PUA prove to be insufficient.

15.8 Noncharacters

Noncharacters: U+FFFE, U+FFFF, and Others

Noncharacters are code points that are permanently reserved in the Unicode Standard for internal use. They are forbidden for use in open interchange of Unicode text data. See *Section 3.4, Characters and Encoding*, for the formal definition of noncharacters and conformance requirements related to their use.

The Unicode Standard sets aside 66 noncharacter code points. The last two code points of each plane are noncharacters: U+FFFE and U+FFFF on the BMP, U+1FFFE and U+1FFFF on Plane 1, and so on, up to U+10FFFE and U+10FFFF on Plane 16, for a total of 34 code points. In addition, there is a contiguous range of another 32 noncharacter code points in the BMP: U+FDD0..U+FDEF. For historical reasons, the range U+FDD0..U+FDEF is contained within the Arabic Presentation Forms-A block, but those noncharacters are not "Arabic noncharacters" or "right-to-left noncharacters," and are not distinguished in any other way from the other noncharacters, except in their code point values.

Applications are free to use any of these noncharacter code points internally but should *never* attempt to exchange them. If a noncharacter is received in open interchange, an application is not required to interpret it in any way. It is good practice, however, to recognize it as a noncharacter and to take appropriate action, such as removing it from the text. Note that Unicode conformance freely allows the removal of these characters. (See C10 in *Section 3.2, Conformance Requirements.*)

In effect, noncharacters can be thought of as application-internal private-use code points. Unlike the private-use characters discussed in *Section 15.7, Private-Use Characters*, which *are* assigned characters and which *are* intended for use in open interchange, subject to interpretation by private agreement, noncharacters are permanently reserved (unassigned) and have no interpretation whatsoever outside of their possible application-internal private uses.

U+FFFF and U+10FFFF. These two noncharacter code points have the attribute of being associated with the largest code unit values for particular Unicode encoding forms. In UTF-16, U+FFFF is associated with the largest 16-bit code unit value, $FFFF_{16}$. U+10FFFF is associated with the largest legal UTF-32 32-bit code unit value, $10FFFF_{16}$. This attribute renders these two noncharacter code points useful for internal purposes as sentinels. For example, they might be used to indicate the end of a list, to represent a value in an index guaranteed to be higher than any valid character value, and so on.

U+FFFE. This noncharacter has the intended peculiarity that, when represented in UTF-16 and then serialized, it has the opposite byte sequence of U+FEFF, the *byte order mark*. This means that applications should reserve U+FFFE as an internal signal that a UTF-16 text stream is in a reversed byte format. Detection of U+FFFE at the start of an input stream should be taken as a strong indication that the input stream should be byte-swapped before interpretation. For more on the use of the *byte order mark* and its interaction with the noncharacter U+FFFE, see *Section 15.9, Specials*.

15.9 Specials

Specials: U+FEFF, U+FFF0–U+FFFD

The Specials block contains code points that are interpreted as neither control nor graphic characters but that are provided to facilitate current software practices. The 14 Unicode code points in the range U+FFF0..U+FFFD are reserved for special character definitions.

For information about the noncharacter code points U+FFFE and U+FFFF, see *Section 15.8, Noncharacters*.

Byte Order Mark (BOM)

There are two circumstances where the character U+FEFF can have a special interpretation that is different from the normal semantics of a *zero width no-break space* (see *Section 15.2, Layout Controls*):

1. Unmarked Byte Order. Some machine architectures use the so-called big-endian byte order, while others use the little-endian byte order. When Unicode text is serialized into bytes, the bytes can go in either order, depending on the architecture. However, sometimes this byte order is not externally marked, which causes problems in interchange between different systems.
2. Unmarked Character Set. In some circumstances, the character set information for a stream of coded characters (such as a file) is not available. The only information available is that the stream contains text, but the precise character set is not known.

In these two cases, the character U+FEFF can be used as a signature to indicate the byte order and the character set by using the byte serializations described in *Section 3.10, Unicode Encoding Schemes*. Because the byte-swapped version U+FFFE is a noncharacter, when an interpreting process finds U+FFFE as the first character, it signals either that the process has encountered text that is of the incorrect byte order or that the file is not valid Unicode text.

In the UTF-16 encoding scheme, U+FEFF at the very beginning of a file or stream explicitly signals the byte order.

The byte sequence <FE$_{16}$ FF$_{16}$> may serve as a signature to identify a file as containing Unicode text. This sequence is exceedingly rare at the outset of text files using other character encodings, whether single- or multiple-byte, and therefore not likely to be confused with real text data. For example, in systems that employ ISO Latin-1 (ISO/IEC 8859-1) or the Microsoft Windows ANSI Code Page 1252, the byte sequence <FE$_{16}$ FF$_{16}$> constitutes the string <*thorn, y diaeresis*> "þÿ"; in systems that employ the Apple Macintosh Roman character set or the Adobe Standard Encoding, this sequence represents the sequence <*ogonek, hacek*> " ˛ ˇ "; in systems that employ other common IBM PC code pages (for example, CP 437, 850), this sequence represents <*black square, no-break space*> "■ ".

In UTF-8, the BOM corresponds to the byte sequence <EF$_{16}$ BB$_{16}$ BF$_{16}$>. Although there are never any questions of byte order with UTF-8 text, this sequence can serve as signature for UTF-8 encoded text where the character set is unmarked. As with a BOM in UTF-16, this sequence of bytes will be extremely rare at the beginning of text files in other character encodings. For example, in systems that employ Microsoft Windows ANSI Code Page 1252, <EF$_{16}$ BB$_{16}$ BF$_{16}$> corresponds to the sequence <*i diaeresis, guillemet, inverted question mark*> "ï » ¿".

For compatibility with versions of the Unicode Standard prior to 3.2, the code point U+FEFF has the word-joining semantics of *zero width no-break space* when it is not used as a byte order mark. In new text these semantics should be encoded by U+2060 WORD JOINER. See "Line and Word Breaking" in *Section 15.2, Layout Controls*, for more information.

Where the byte order is explicitly specified, such as in UTF-16BE or UTF-16LE, then all U+FEFF characters, even at the very beginning of the text, are to be interpreted as *zero width no-break spaces*. Similarly, where Unicode text has known byte order, initial U+FEFF characters are also not required, but for backward compatibility are to be interpreted as *zero width no-break spaces*. For example, for strings in an API, the memory architecture of the processor provides the explicit byte order. For databases and similar structures, it is much more efficient and robust to use a uniform byte order for the same field (if not the entire database), thereby avoiding use of the *byte order mark*.

Systems that use the *byte order mark* must recognize when an initial U+FEFF signals the byte order. In those cases, it is not part of the textual content and should be removed before processing, because otherwise it may be mistaken for a legitimate *zero width no-break space*. To represent an initial U+FEFF ZERO WIDTH NO-BREAK SPACE in a UTF-16 file, use U+FEFF twice in a row. The first one is a *byte order mark*; the second one is the initial *zero width no-break space*. See *Table 15-3* for a summary of encoding form signatures.

Table 15-3. Unicode Encoding Form Signatures

Encoding Form	Signature
UTF-8	EF BB BF
UTF-16 Big-endian	FE FF
UTF-16 Little-endian	FF FE
UTF-32 Big-endian	00 00 FE FF
UTF-32 Little-endian	FF FE 00 00

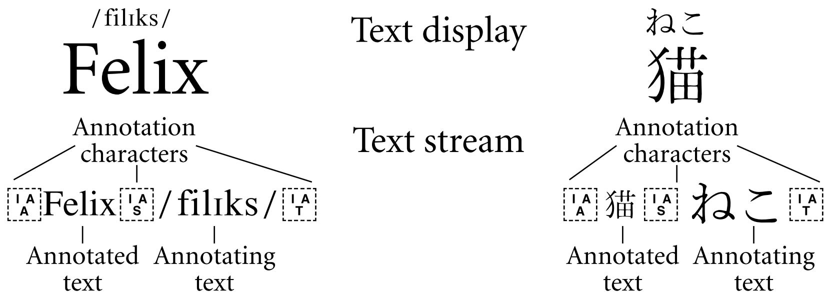
If U+FEFF had only the semantic of a signature code point, it could be freely deleted from text without affecting the interpretation of the rest of the text. Carelessly appending files together, for example, can result in a signature code point in the middle of text. Unfortunately, U+FEFF also has significance as a character. As a *zero width no-break space*, it indicates that line breaks are not allowed between the adjoining characters. Thus U+FEFF affects the interpretation of text and cannot be freely deleted. The overloading of semantics for this code point has caused problems for programs and protocols. The new character U+2060 WORD JOINER has the same semantics in all cases as U+FEFF, except that it *cannot* be used as a signature. Implementers are strongly encouraged to use word joiner in those circumstances whenever word joining semantics is intended.

Annotation Characters

An *interlinear annotation* consists of *annotating text* that is related to a sequence of *annotated* characters. For all regular editing and text-processing algorithms, the annotated characters are treated as part of the text stream. The annotating text is also part of the content, but for all or some text processing, it does not form part of the main text stream. However, within the annotating text, characters are accessible to the same kind of layout, text-processing, and editing algorithms as the base text. The *annotation character*s delimit the annotating and the annotated text, and identify them as part of an annotation. See *Figure 15-3*.

The annotation characters are used in internal processing when out-of-band information is associated with a character stream, very similarly to the usage of U+FFFC OBJECT REPLACE-

Figure 15-3. Annotation Characters



MENT CHARACTER. However, unlike the opaque objects hidden by the latter character, the annotation itself is textual.

Conformance. A conformant implementation that supports annotation characters interprets the base text as if it were part of an unannotated text stream. Within the annotating text, it interprets the annotating characters with their regular Unicode semantics.

U+FFF9 INTERLINEAR ANNOTATION ANCHOR is an anchor character, preceding the interlinear annotation. The exact nature and formatting of the annotation is dependent on additional information that is not part of the plain text stream. This situation is analogous to that for U+FFFC OBJECT REPLACEMENT CHARACTER.

U+FFFA INTERLINEAR ANNOTATION SEPARATOR separates the base characters in the text stream from the annotation characters that follow. The exact interpretation of this character depends on the nature of the annotation. More than one separator may be present. Additional separators delimit parts of a multipart annotating text.

U+FFFB INTERLINEAR ANNOTATION TERMINATOR terminates the annotation object (and returns to the regular text stream).

Use in Plain Text. Usage of the annotation characters in plain text interchange is strongly discouraged without prior agreement between the sender and the receiver, because the content may be misinterpreted otherwise. Simply filtering out the annotation characters on input will produce an unreadable result or, even worse, an opposite meaning. On input, a plain text receiver should either preserve all characters or remove the interlinear annotation characters as well as the annotating text included between the INTERLINEAR ANNOTATION SEPARATOR and the INTERLINEAR ANNOTATION TERMINATOR.

When an output for plain text usage is desired and when the receiver is unknown to the sender, these interlinear annotation characters should be removed as well as the annotating text included between the INTERLINEAR ANNOTATION SEPARATOR and the INTERLINEAR ANNOTATION TERMINATOR.

This restriction does not preclude the use of annotation characters in plain text interchange, but it requires a prior agreement between the sender and the receiver for correct interpretation of the annotations.

Lexical Restrictions. If an implementation encounters a paragraph break between an *anchor* and its corresponding *terminator,* it shall terminate any open annotations at this point. Anchor characters must precede their corresponding terminator characters. Unpaired anchors or terminators shall be ignored. A *separator* occurring outside a pair of delimiters, shall be ignored. Annotations may be nested.

Formatting. All formatting information for an annotation is provided by higher-level protocols. The details of the layout of the annotation are implementation-defined. Correct formatting may require additional information not present in the character stream, but maintained out-of-band. Therefore, annotation markers serve as placeholders for an implementation that has access to that information from another source.

Collation. With the exception of the special case where the annotation is intended to be used as a sortkey, annotations are typically ignored for collation or optionally preprocessed to act as tie breakers only. Importantly, annotation base characters are not ignored, but rather are treated like regular text.

Replacement Characters

U+FFFD. The U+FFFD REPLACEMENT CHARACTER is the general substitute character in the Unicode Standard. It can be substituted for any "unknown" character in another encoding that cannot be mapped in terms of known Unicode characters (see *Section 5.3, Unknown and Missing Characters*).

U+FFFC. The U+FFFC OBJECT REPLACEMENT CHARACTER is used as an insertion point for objects located within a stream of text. All other information about the object is kept outside the character data stream. Internally it is a dummy character that acts as an anchor point for the object's formatting information. In addition to assuring correct placement of an object in a data stream, the object replacement character allows the use of general stream-based algorithms for any textual aspects of embedded objects.

15.10 Tag Characters

Tag Characters: U+E0000–U+E007F

The characters in this block provide a mechanism for language tagging in Unicode plain text. *The use of these characters is strongly discouraged.* The characters in this block are reserved for use with special protocols. They are *not* to be used in the absence of such protocols or with *any* protocols that provide alternate means for language tagging, such as HTML or XML. See Unicode Technical Report #20, "Unicode in XML and Other Markup Languages." The requirement for language information embedded in plain text data is often overstated. See *Section 5.10, Language Information in Plain Text*.

This block encodes a set of 95 special-use tag characters to enable the spelling out of ASCII-based string tags using characters which can be strictly separated from ordinary text content characters in Unicode. These tag characters can be embedded by protocols into plain text. They can be identified and/or ignored by implementations with trivial algorithms because there is no overloading of usage for these tag characters—they can only express tag values and never textual content itself.

In addition to these 95 characters, one language tag identification character and one cancel tag character are encoded. The language tag identification character identifies a tag string as a language tag; the language tag itself makes use of RFC 3066 (or its successors) language tag strings spelled out using the tag characters from this block.

Syntax for Embedding Tags

To embed any ASCII-derived tag in Unicode plain text, the tag is spelled out with corresponding tag characters, prefixed with the relevant tag identification character. The resultant string is embedded directly in the text.

Tag Identification. The tag identification character is used as a mechanism for identifying tags of different types. In the future, this could enable multiple types of tags embedded in plain text to coexist.

Tag Termination. No termination character is required for the tag itself, because all characters that make up the tag are numerically distinct from any non-tag character. A tag terminates either at the first non-tag character (that is, any other normal Unicode character) or at next tag identification character. A detailed BNF syntax for tags is listed in "Formal Tag Syntax" later in this section.

Language Tags. A string of tag characters prefixed by U+E0001 LANGUAGE TAG is specified to constitute a language tag. Furthermore, the tag values for the language tag are to be spelled out as specified in RFC 3066, making use only of registered tag values or of user-defined language tags starting with the characters "x-".

For example, consider the task of embedding a language tag for Japanese. The Japanese tag from RFC 3066 is "ja" (composed of ISO 639 language id) or, alternatively, "ja-JP" (composed of ISO 639 language id plus ISO 3166 country id). Because RFC 3066 specifies that language tags are not case significant, it is recommended that for language tags, the entire tag be lowercased before conversion to tag characters.

Thus the entire language tag "ja-JP" would be converted to the tag characters as follows:

<U+E0001, U+E006A, U+E0061, U+E002D, U+E006A, U+E0070>

The language tag, in its shorter, "ja" form, would be expressed as follows:

<U+E0001, U+E006A, U+E0061>

Tag Scope and Nesting. The value of an established tag continues from the point at which the tag is embedded in text until either

> A. The text itself goes out of scope, as defined by the application, for example, for line-oriented protocols, when reaching the end-of-line or end-of-string; for text streams, when reaching the end-of-stream; and so on),

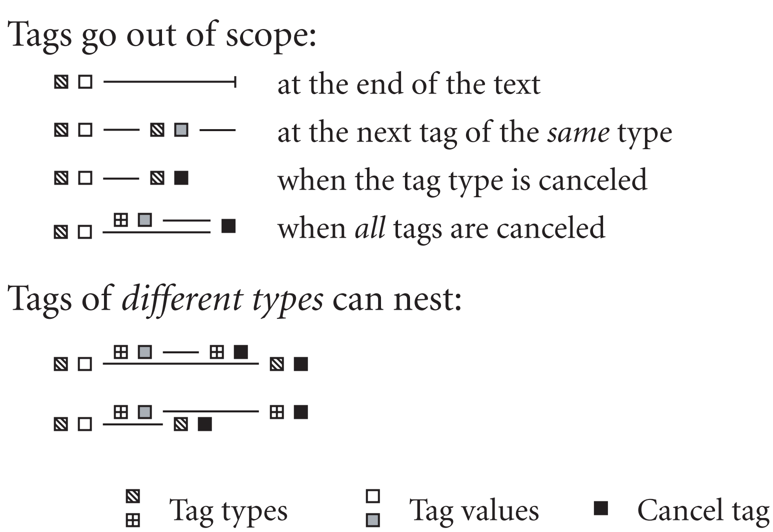
or

> B. The tag is explicitly canceled by the U+E007F CANCEL TAG character.

Tags of the *same* type cannot be nested in any way. For example, if a new embedded language tag occurs following text that was already language tagged, the tagged value for subsequent text simply changes to that specified in the new tag.

Tags of different types can have interdigitating scope, but not hierarchical scope. In effect, tags of different types completely ignore each other, so that the use of language tags can be completely asynchronous with the use of future tag types. These relationships are illustrated in *Figure 15-4*.

Figure 15-4. Tag Characters



Canceling Tag Values. The main function of CANCEL TAG is to make possible operations such as blind concatenation of strings in a tagged context without the propagation of inappropriate tag values across the string boundaries. There are two uses of CANCEL TAG. To cancel a tag value of a particular type, prefix the CANCEL TAG character with the tag identification character of the appropriate type. For example, the complete string to cancel a language tag is <U+E0001, U+E007F>.

The value of the relevant tag type returns to the default state for that tag type—namely, no tag value specified, the same as untagged text. To cancel any tag values of any type that may be in effect, use CANCEL TAG without a prefixed tag identification character.

Currently there is no observable difference in the two uses of CANCEL TAG, because only one tag identification character (and therefore one tag type) is defined. Inserting a bare CANCEL TAG in places where only the language tag needs to be canceled could lead to unanticipated side effects if this text were to be inserted in the future into a text that supports more than one tag type.

Working with Language Tags

Avoiding Language Tags. Because of the extra implementation burden, language tags should be avoided in plain text unless language information is required and it is known that the receivers of the text will properly recognize and maintain the tags. However, where language tags must be used, implementers should consider the following implementation issues involved in supporting language information with tags and decide how to handle tags where they are not fully supported. This discussion applies to any mechanism for providing language tags in a plain text environment.

Higher-Level Protocols. Language tags should be avoided wherever higher-level protocols, such as a rich text format, HTML, or MIME, provide language attributes. This practice prevents cases where the higher-level protocol and the language tags disagree. See Unicode Technical Report #20, "Unicode in XML and Other Markup Languages."

Effect of Tags on Interpretation of Text. Implementations that support language tags may need to take them into account for special processing, such as hyphenation or choice of font. However, the tag characters themselves have no display and do not affect line breaking, character shaping or joining, or any other format or layout properties. Processes interpreting the tag may choose to impose such behavior based on the tag value that it represents.

Display. Characters in the tag character block have no visible rendering in normal text and the language tags themselves are not displayed. This choice may not require modification of the displaying program, if the fonts on that platform have the language tag characters mapped to zero-width, invisible glyphs. For debugging or other operations that must render the tags themselves visible, it is advisable that the tag characters be rendered using the corresponding ASCII character glyphs (perhaps modified systematically to differentiate them from normal ASCII characters). But the tag character values are chosen so that the tag characters will be interpretable in most debuggers even without display support.

Processing. Sequential access to the text is generally straightforward. If language codes are not relevant to the particular processing operation, then they should be ignored. Random access to stateful tags is more problematic. Because the current state of the text depends upon tags previous to it, the text must be searched backward, sometimes all the way to the start. With these exceptions, tags pose no particular difficulties as long as no modifications are made to the text.

Range Checking for Tag Characters. Tag characters are encoded in Plane 14 to support easy range checking. The following C/C++ source code snippets show efficient implementations of range checks for characters E0000 to E007F expressed in each of the three significant Unicode encoding forms. Range checks allow implementations that do not want to support these tag characters to efficiently filter for them.

Range check expressed in UTF-32:

```
if ( ((unsigned)  *s)  - 0xE0000  <= 0x7F  )
```

Range check expressed in UTF-16:

```
if ( ( *s == 0xDB40 ) && ( ((unsigned)*(s+1)) - 0xDC00  <= 0x7F ) )
```

Range check expressed in UTF-8:

```
if ( ( *s == 0xF3 ) && ( *(s+1) == 0xA0 ) &&
        ( ( *(s+2) & 0xFE ) == 0x80 ) )
```

Alternatively, the range checks for UTF-32 and UTF-16 can be coded with bit masks. Both versions should be equally efficient.

Range check expressed in UTF-32:

```
if ( ((*s) & 0xFFFFFF80)  == 0xE0000 )
```

Range check expressed in UTF-16:

```
if ( ( *s == 0xDB40 ) && ( *(s+1) & 0xDC80) == 0xDC00 )
```

Editing and Modification. Inline tags present particular problems for text changes, because they are stateful. Any modifications of the text are more complicated, as those modifications need to be aware of the current language status and the <start>...<end> tags must be properly maintained. If an editing program is unaware that certain tags are stateful and cannot process them correctly, then it is very easy for the user to modify text in ways that corrupt it. For example, a user might delete part of a tag or paste text including a tag into the wrong context.

Dangers of Incomplete Support. Even programs that do not interpret the tags should not allow editing operations to break initial tags or leave tags unpaired. Unpaired tags should be discarded upon a save or send operation.

Nonetheless, malformed text may be produced and transmitted by a tag-unaware editor. Therefore, implementations that do not ignore language tags must be prepared to receive malformed tags. On reception of a malformed or unpaired tag, language tag-aware implementations should reset the language to NONE, and then ignore the tag.

Unicode Conformance Issues

The rules for Unicode conformance for the tag characters are exactly the same as those for any other Unicode characters. A conformant process is not required to interpret the tag characters. If it does interpret them, it should interpret them according to the standard—that is, as spelled-out tags. However, there is no requirement to provide a particular interpretation of the text because it is tagged with a given language. If an application does not interpret tag characters, it should leave their values undisturbed and do whatever it does with any other uninterpreted characters.

The presence of a well-formed tag is no guarantee that the data is correctly tagged. For example, an application could erroneously label French data with a Spanish tag.

Implementations of Unicode that already make use of out-of-band mechanisms for language tagging or "heavy-weight" in-band mechanisms such as XML or HTML will continue to do exactly what they are doing and will ignore the tag characters completely, and may prohibit their use to prevent conflicts with the equivalent markup.

Formal Tag Syntax

An extended BNF description of the tags specified in this section is given here.

```
tag := language-tag | cancel-all-tag

language-tag := language-tag-introducer (language-tag-arg
                | tag-cancel)

language-tag-arg := tag-argument
```

> In this rule, `tag-argument` is constrained to be a valid language identifier according to RFC 3066, with the assumption that the appropriate conversions from tag character values to ASCII are performed, before

checking for syntactic correctness against RFC 3066. For example, U+E0041 TAG LATIN CAPITAL LETTER A is mapped to U+0041 LATIN CAPITAL LETTER A, and so on.

```
cancel-all-tag := tag-cancel

tag-argument := tag-character+

tag-character := [U+E0020 - U+E007E]

language-tag-introducer := U+E0001

tag-cancel := U+E007F
```