The Unicode® Standard
Version 9.0 – Core Specification

To learn about the latest version of the Unicode Standard, see http://www.unicode.org/versions/latest/.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

Unicode and the Unicode Logo are registered trademarks of Unicode, Inc., in the United States and other countries.

The authors and publisher have taken care in the preparation of this specification, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The *Unicode Character Database* and other files are provided as-is by Unicode, Inc. No claims are made as to fitness for any particular purpose. No warranties of any kind are expressed or implied. The recipient agrees to determine applicability of information provided.

Chapter 5

Implementation Guidelines

It is possible to implement a substantial subset of the Unicode Standard as "wide ASCII" with little change to existing programming practice. However, the Unicode Standard also provides for languages and writing systems that have more complex behavior than English does. Whether one is implementing a new operating system from the ground up or enhancing existing programming environments or applications, it is necessary to examine many aspects of current programming practice and conventions to deal with this more complex behavior.

This chapter covers a series of short, self-contained topics that are useful for implementers. The information and examples presented here are meant to help implementers understand and apply the design and features of the Unicode Standard. That is, they are meant to promote good practice in implementations conforming to the Unicode Standard.

These recommended guidelines are not normative and are not binding on the implementer, but are intended to represent best practice. When implementing the Unicode Standard, it is important to look not only at the letter of the conformance rules, but also at their spirit. Many of the following guidelines have been created specifically to assist people who run into issues with conformant implementations, while reflecting the requirements of actual usage.

5.1 Data Structures for Character Conversion

The Unicode Standard exists in a world of other text and character encoding standards—some private, some national, some international. A major strength of the Unicode Standard is the number of other important standards that it incorporates. In many cases, the Unicode Standard included duplicate characters to guarantee round-trip transcoding to established and widely used standards.

Issues

Conversion of characters between standards is not always a straightforward proposition. Many characters have mixed semantics in one standard and may correspond to more than one character in another. Sometimes standards give duplicate encodings for the same character; at other times the interpretation of a whole set of characters may depend on the application. Finally, there are subtle differences in what a standard may consider a character.

For these reasons, mapping tables are usually required to map between the Unicode Standard and another standard. Mapping tables need to be used consistently for text data exchange to avoid modification and loss of text data. For details, see Unicode Technical Standard #22, "Character Mapping Markup Language (CharMapML)." By contrast, conversions between different Unicode encoding forms are fast, lossless permutations.

There are important security issues associated with encoding conversion. For more information, see Unicode Technical Report #36, "Unicode Security Considerations."

The Unicode Standard can be used as a pivot to transcode among n different standards. This process, which is sometimes called *triangulation*, reduces the number of mapping tables that an implementation needs from $O(n^2)$ to $O(n)$.

Multistage Tables

Tables require space. Even small character sets often map to characters from several different blocks in the Unicode Standard and thus may contain up to 64K entries (for the BMP) or 1,088K entries (for the entire codespace) in at least one direction. Several techniques exist to reduce the memory space requirements for mapping tables. These techniques apply not only to transcoding tables, but also to many other tables needed to implement the Unicode Standard, including character property data, case mapping, collation tables, and glyph selection tables.

Flat Tables. If diskspace is not at issue, virtual memory architectures yield acceptable working set sizes even for flat tables because the frequency of usage among characters differs widely. Even small character sets contain many infrequently used characters. In addition, data intended to be mapped into a given character set generally does not contain characters from all blocks of the Unicode Standard (usually, only a few blocks at a time need to be transcoded to a given character set). This situation leaves certain sections of the mapping tables unused—and therefore paged to disk. The effect is most pronounced for large tables mapping from the Unicode Standard to other character sets, which have large

sections simply containing mappings to the default character, or the "unmappable character" entry.

Ranges. It may be tempting to "optimize" these tables for space by providing elaborate provisions for nested ranges or similar devices. This practice leads to unnecessary performance costs on modern, highly pipelined processor architectures because of branch penalties. A faster solution is to use an *optimized two-stage table*, which can be coded without any test or branch instructions. Hash tables can also be used for space optimization, although they are not as fast as multistage tables.

Two-Stage Tables. Two-stage tables are a commonly employed mechanism to reduce table size (see *Figure 5-1*). They use an array of pointers and a default value. If a pointer is NULL, the value returned by a lookup operation in the table is the default value. Otherwise, the pointer references a block of values used for the second stage of the lookup. For BMP characters, it is quite efficient to organize such two-stage tables in terms of high byte and low byte values. The first stage is an array of 256 pointers, and each of the secondary blocks contains 256 values indexed by the low byte in the code point. For supplementary characters, it is often advisable to structure the pointers and second-stage arrays somewhat differently, so as to take best advantage of the very sparse distribution of supplementary characters in the remaining codespace.

Figure 5-1. Two-Stage Tables
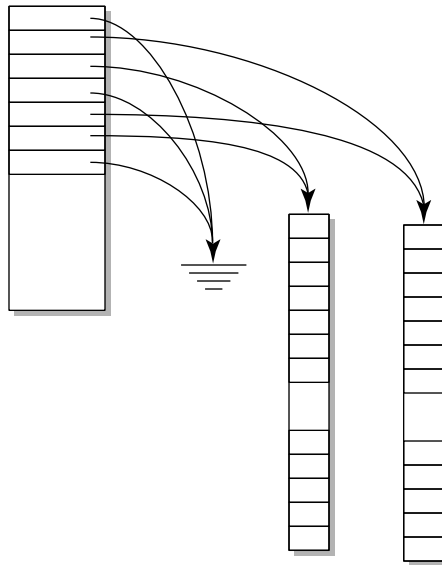
Optimized Two-Stage Table. Wherever any blocks are identical, the pointers just point to the same block. For transcoding tables, this case occurs generally for a block containing only mappings to the default or "unmappable" character. Instead of using NULL pointers and a default value, one "shared" block of default entries is created. This block is pointed to

by all first-stage table entries, for which no character value can be mapped. By avoiding tests and branches, this strategy provides access time that approaches the simple array access, but at a great savings in storage.

Multistage Table Tuning. Given a table of arbitrary size and content, it is a relatively simple matter to write a small utility that can calculate the optimal number of stages and their width for a multistage table. Tuning the number of stages and the width of their arrays of index pointers can result in various trade-offs of table size versus average access time.

5.2 Programming Languages and Data Types

Programming languages provide for the representation and handling of characters and strings via data types, data constants (literals), and methods. Explicit support for Unicode helps with the development of multilingual applications. In some programming languages, strings are expressed as sequences (arrays) of primitive types, exactly corresponding to sequences of code units of one of the Unicode encoding forms. In other languages, strings are objects, but indexing into strings follows the semantics of addressing code units of a particular encoding form.

Data types for "characters" generally hold just a single Unicode code point value for low-level processing and lookup of character property values. When a primitive data type is used for single-code point values, a *signed* integer type can be useful; negative values can hold "sentinel" values like end-of-string or end-of-file, which can be easily distinguished from Unicode code point values. However, in most APIs, string types should be used to accommodate user-perceived characters, which may require sequences of code points.

Unicode Data Types for C

ISO/IEC Technical Report 19769, *Extensions for the programming language C to support new character types*, defines data types for the three Unicode encoding forms (UTF-8, UTF-16, and UTF-32), syntax for Unicode string and character literals, and methods for the conversion between the Unicode encoding forms. No other methods are specified.

Unicode strings are encoded as arrays of primitive types as usual. For UTF-8, UTF-16, and UTF-32, the basic types are char, char16_t, and char32_t, respectively. The ISO Technical Report assumes that char is at least 8 bits wide for use with UTF-8. While char and wchar_t may be signed or unsigned types, the new char16_t and char32_t types are defined to be unsigned integer types.

Unlike the specification in the wchar_t programming model, the Unicode data types do not require that a single string base unit alone (especially char or char16_t) must be able to store any one character (code point).

UTF-16 string and character literals are written with a lowercase u as a prefix, similar to the L prefix for wchar_t literals. UTF-32 literals are written with an uppercase U as a prefix. Characters outside the basic character set are available for use in string literals through the \uhhhh and \Uhhhhhhhh escape sequences.

These types and semantics are available in a compiler if the <uchar.h> header is present and defines the __STDC_UTF_16__ (for char16_t) and __STDC_UTF_32__ (for char32_t) macros.

Because Technical Report 19769 was not available when UTF-16 was first introduced, many implementations have been supporting a 16-bit wchar_t to contain UTF-16 code units. Such usage is not conformant to the C standard, because supplementary characters require use of pairs of wchar_t units in this case.

ANSI/ISO C wchar_t. With the `wchar_t` wide character type, ANSI/ISO C provides for inclusion of fixed-width, wide characters. ANSI/ISO C leaves the semantics of the wide character set to the specific implementation but requires that the characters from the portable C execution set correspond to their wide character equivalents by zero extension. The Unicode characters in the ASCII range U+0020 to U+007E satisfy these conditions. Thus, if an implementation uses ASCII to code the portable C execution set, the use of the Unicode character set for the `wchar_t` type, in either UTF-16 or UTF-32 form, fulfills the requirement.

The width of `wchar_t` is compiler-specific and can be as small as 8 bits. Consequently, programs that need to be portable across any C or C++ compiler should not use `wchar_t` for storing Unicode text. The `wchar_t` type is intended for storing compiler-defined wide characters, which may be Unicode characters in some compilers. However, programmers who want a UTF-16 implementation can use a macro or typedef (for example, `UNICHAR`) that can be compiled as `unsigned short` or `wchar_t` depending on the target compiler and platform. Other programmers who want a UTF-32 implementation can use a macro or typedef that might be compiled as `unsigned int` or `wchar_t`, depending on the target compiler and platform. This choice enables correct compilation on different platforms and compilers. Where a 16-bit implementation of `wchar_t` is guaranteed, such macros or typedefs may be predefined (for example, `TCHAR` on the Win32 API).

On systems where the native character type or `wchar_t` is implemented as a 32-bit quantity, an implementation may use the UTF-32 form to represent Unicode characters.

A limitation of the ISO/ANSI C model is its assumption that characters can always be processed in isolation. Implementations that choose to go beyond the ISO/ANSI C model may find it useful to mix widths within their APIs. For example, an implementation may have a 32-bit `wchar_t` and process strings in any of the UTF-8, UTF-16, or UTF-32 forms. Another implementation may have a 16-bit `wchar_t` and process strings as UTF-8 or UTF-16, but have additional APIs that process individual characters as UTF-32 or deal with pairs of UTF-16 code units.

5.3 Unknown and Missing Characters

This section briefly discusses how users or implementers might deal with characters that are not supported or that, although supported, are unavailable for legible rendering.

Reserved and Private-Use Character Codes. There are two classes of code points that even a "complete" implementation of the Unicode Standard cannot necessarily interpret correctly:

 • Code points that are reserved

 • Code points in the Private Use Area for which no private agreement exists

An implementation should not attempt to interpret such code points. However, in practice, applications must deal with unassigned code points or private-use characters. This may occur, for example, when the application is handling text that originated on a system implementing a later release of the Unicode Standard, with additional assigned characters.

Options for rendering such unknown code points include printing the code point as four to six hexadecimal digits, printing a black or white box, using appropriate glyphs such as ⌧ for reserved and ⌀ for private use, or simply displaying nothing. An implementation should not blindly delete such characters, nor should it unintentionally transform them into something else.

Interpretable but Unrenderable Characters. An implementation may receive a code point that is assigned to a character in the Unicode character encoding, but be unable to render it because it lacks a font for the code point or is otherwise incapable of rendering it appropriately.

In this case, an implementation might be able to provide limited feedback to the user's queries, such as being able to sort the data properly, show its script, or otherwise display the code point in a default manner. An implementation can distinguish between unrenderable (but assigned) code points and unassigned code points by printing the former with distinctive glyphs that give some general indication of their type, such as Ⓛ, Ⓡ, Ⓞ, ⓞ, ⓐ, ⑤, ⓙ, ⓗ, ⓣ, ⓖ, 漢, and so on.

Default Ignorable Code Points. Normally, characters outside the repertoire of supported characters for an implementation would be graphical characters displayed with a fallback glyph, such as a black box. However, certain special-use characters, such as format controls or variation selectors, do not have visible glyphs of their own, although they may have an effect on the display of other characters. When such a special-use character is not supported by an implementation, it should not be displayed with a visible fallback glyph, but instead simply not be rendered at all. The list of such characters which should not be rendered with a fallback glyph is defined by the Default_Ignorable_Code_Point property in the Unicode Character Database. For more information, see *Section 5.21, Ignoring Characters in Processing*.

Interacting with Downlevel Systems. Versions of the Unicode Standard after Unicode 2.0 are strict supersets of Unicode 2.0 and all intervening versions. The Derived Age property

tracks the version of the standard at which a particular character was added to the standard. This information can be particularly helpful in some interactions with downlevel systems. If the protocol used for communication between the systems provides for an announcement of the Unicode version on each one, an uplevel system can predict which recently added characters will appear as unassigned characters to the downlevel system.

5.4 Handling Surrogate Pairs in UTF-16

The method used by UTF-16 to address the 1,048,576 supplementary code points that cannot be represented by a single 16-bit value is called *surrogate pairs*. A surrogate pair consists of a high-surrogate code unit (leading surrogate) followed by a low-surrogate code unit (trailing surrogate), as described in the specifications in *Section 3.8, Surrogates*, and the UTF-16 portion of *Section 3.9, Unicode Encoding Forms*.

In well-formed UTF-16, a trailing surrogate can be preceded only by a leading surrogate and not by another trailing surrogate, a non-surrogate, or the start of text. A leading surrogate can be followed only by a trailing surrogate and not by another leading surrogate, a non-surrogate, or the end of text. Maintaining the well-formedness of a UTF-16 code sequence or accessing characters within a UTF-16 code sequence therefore puts additional requirements on some text processes. Surrogate pairs are designed to minimize this impact.

Leading surrogates and trailing surrogates are assigned to disjoint ranges of code units. In UTF-16, non-surrogate code points can never be represented with code unit values in those ranges. Because the ranges are disjoint, each code unit in well-formed UTF-16 must meet one of only three possible conditions:

- A single non-surrogate code unit, representing a code point between 0 and $D7FF_{16}$ or between $E000_{16}$ and $FFFF_{16}$
- A leading surrogate, representing the first part of a surrogate pair
- A trailing surrogate, representing the second part of a surrogate pair

By accessing at most two code units, a process using the UTF-16 encoding form can therefore interpret any Unicode character. Determining character boundaries requires at most scanning one preceding or one following code unit without regard to any other context.

As long as an implementation does not remove either of a pair of surrogate code units or incorrectly insert another character between them, the integrity of the data is maintained. Moreover, even if the data becomes corrupted, the corruption remains localized, unlike with some other multibyte encodings such as Shift-JIS or EUC. Corrupting a single UTF-16 code unit affects only a single character. Because of non-overlap (see *Section 2.5, Encoding Forms*), this kind of error does not propagate throughout the rest of the text.

UTF-16 enjoys a beneficial frequency distribution in that, for the majority of all text data, surrogate pairs will be very rare; non-surrogate code points, by contrast, will be very common. Not only does this help to limit the performance penalty incurred when handling a variable-width encoding, but it also allows many processes either to take no specific action for surrogates or to handle surrogate pairs with existing mechanisms that are already needed to handle character sequences.

Implementations should fully support surrogate pairs in processing UTF-16 text. Without surrogate support, an implementation would not interpret any supplementary characters or guarantee the integrity of surrogate pairs. This might apply, for example, to an older implementation, conformant to Unicode Version 1.1 or earlier, before UTF-16 was defined.

Support for supplementary characters is important because a significant number of them are relevant for modern use, despite their low frequency.

The individual *components* of implementations may have different levels of support for surrogates, as long as those components are assembled and communicate correctly. Low-level string processing, where a Unicode string is not interpreted but is handled simply as an array of code units, may ignore surrogate pairs. With such strings, for example, a truncation operation with an arbitrary offset might break a surrogate pair. (For further discussion, see *Section 2.7, Unicode Strings*.) For performance in string operations, such behavior is reasonable at a low level, but it requires higher-level processes to ensure that offsets are on character boundaries so as to guarantee the integrity of surrogate pairs.

Strategies for Surrogate Pair Support. Many implementations that handle advanced features of the Unicode Standard can easily be modified to support surrogate pairs in UTF-16. For example:

- Text collation can be handled by treating those surrogate pairs as "grouped characters," such as is done for "ij" in Dutch or "ch" in Slovak.

- Text entry can be handled by having a keyboard generate two Unicode code points with a single keypress, much as an ENTER key can generate CRLF or an Arabic keyboard can have a "*lam-alef*" key that generates a sequence of two characters, *lam* and *alef*.

- Truncation can be handled with the same mechanism as used to keep combining marks with base characters. For more information, see Unicode Standard Annex #29, "Unicode Text Segmentation."

Users are prevented from damaging the text if a text editor keeps *insertion points* (also known as *carets*) on character boundaries.

Implementations using UTF-8 and Unicode 8-bit strings necessitate similar considerations. The main difference from handling UTF-16 is that in the UTF-8 case the only characters that are represented with single code units (single bytes) in UTF-8 are the ASCII characters, U+0000..U+007F. Characters represented with multibyte sequences are very common in UTF-8, unlike surrogate pairs in UTF-16, which are rather uncommon. This difference in frequency may result in different strategies for handling the multibyte sequences in UTF-8.

5.5 Handling Numbers

There are many sets of characters that represent decimal digits in different scripts. Systems that interpret those characters numerically should provide the correct numerical values. For example, the sequence <U+0968 DEVANAGARI DIGIT TWO, U+0966 DEVANAGARI DIGIT ZERO> when numerically interpreted has the value *twenty.*

When converting binary numerical values to a visual form, digits can be chosen from different scripts. For example, the value *twenty* can be represented either by <U+0032 DIGIT TWO, U+0030 DIGIT ZERO> or by <U+0968 DEVANAGARI DIGIT TWO, U+0966 DEVANAGARI DIGIT ZERO> or by <U+0662 ARABIC-INDIC DIGIT TWO, U+0660 ARABIC-INDIC DIGIT ZERO>. It is recommended that systems allow users to choose the format of the resulting digits by replacing the appropriate occurrence of U+0030 DIGIT ZERO with U+0660 ARABIC-INDIC DIGIT ZERO, and so on. (See *Chapter 4, Character Properties,* for the information needed to implement formatting and scanning numerical values.)

Fullwidth variants of the ASCII digits are simply compatibility variants of regular digits and should be treated as regular Western digits.

The Roman numerals, Greek acrophonic numerals, and East Asian ideographic numerals are decimal numeral writing systems, but they are not formally decimal radix digit systems. That is, it is not possible to do a one-to-one transcoding to forms such as 123456.789. Such systems are appropriate only for positive integer writing.

It is also possible to write numbers in two ways with CJK ideographic digits. For example, *Figure 22-6* shows how the number 1,234 can be written. Supporting these ideographic digits for numerical parsing means that implementations must be smart about distinguishing between the two cases.

Digits often occur in situations where they need to be parsed, but are not part of numbers. One such example is alphanumeric identifiers (see Unicode Standard Annex #31, "Unicode Identifier and Pattern Syntax").

Only in higher-level protocols, such as when implementing a full mathematical formula parser, do considerations such as superscripting and subscripting of digits become crucial for numerical interpretation.

5.6 Normalization

Alternative Spellings. The Unicode Standard contains explicit codes for the most frequently used accented characters. These characters can also be composed; in the case of accented letters, characters can be composed from a base character and nonspacing mark(s).

The Unicode Standard provides decompositions for characters that can be composed using a base character plus one or more nonspacing marks. The decomposition mappings are specific to a particular version of the Unicode Standard. Further decomposition mappings may be added to the standard for new characters encoded in the future; however, no existing decomposition mapping for a currently encoded character will ever be removed or changed, nor will a decomposition mapping be added for a currently encoded character. These constraints on changes for decomposition are enforced by the Normalization Stability Policy. See the subsection "Policies" in *Section B.6, Other Unicode Online Resources*.

Normalization. Systems may normalize Unicode-encoded text to one particular sequence, such as normalizing composite character sequences into precomposed characters, or vice versa (see *Figure 5-2*).

Figure 5-2. Normalization
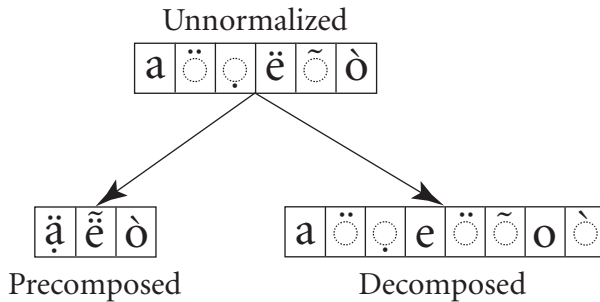
Compared to the number of *possible* combinations, only a relatively small number of precomposed base character plus nonspacing marks have independent Unicode character values.

Systems that cannot handle nonspacing marks can normalize to precomposed characters; this option can accommodate most modern Latin-based languages. Such systems can use fallback rendering techniques to at least visually indicate combinations that they cannot handle (see the "Fallback Rendering" subsection of *Section 5.13, Rendering Nonspacing Marks*).

In systems that *can* handle nonspacing marks, it may be useful to normalize so as to eliminate precomposed characters. This approach allows such systems to have a homogeneous representation of composed characters and maintain a consistent treatment of such char-

acters. However, in most cases, it does not require too much extra work to support mixed forms, which is the simpler route.

The Unicode Normalization Forms are defined in *Section 3.11, Normalization Forms*. For further information about implementation of normalization, see also Unicode Standard Annex #15, "Unicode Normalization Forms." For a general discussion of issues related to normalization, see "Equivalent Sequences" in *Section 2.2, Unicode Design Principles*; and *Section 2.11, Combining Characters*.

5.7 Compression

Using the Unicode character encoding may increase the amount of storage or memory space dedicated to the text portion of files. Compressing Unicode-encoded files or strings can therefore be an attractive option if the text portion is a large part of the volume of data compared to binary and numeric data, and if the processing overhead of the compression and decompression is acceptable.

Compression always constitutes a higher-level protocol and makes interchange dependent on knowledge of the compression method employed. For a detailed discussion of compression and a standard compression scheme for Unicode, see Unicode Technical Standard #6, "A Standard Compression Scheme for Unicode."

Encoding forms defined in *Section 2.5, Encoding Forms*, have different storage characteristics. For example, as long as text contains only characters from the Basic Latin (ASCII) block, it occupies the same amount of space whether it is encoded with the UTF-8 or ASCII codes. Conversely, text consisting of CJK ideographs encoded with UTF-8 will require more space than equivalent text encoded with UTF-16.

For processing rather than storage, the Unicode encoding form is usually selected for easy interoperability with existing APIs. Where there is a choice, the trade-off between decoding complexity (high for UTF-8, low for UTF-16, trivial for UTF-32) and memory and cache bandwidth (high for UTF-32, low for UTF-8 or UTF-16) should be considered.

5.8 Newline Guidelines

Newlines are represented on different platforms by carriage return (CR), line feed (LF), CRLF, or next line (NEL). Not only are newlines represented by different characters on different platforms, but they also have ambiguous behavior even on the same platform. These characters are often transcoded directly into the corresponding Unicode code points when a character set is transcoded; this means that even programs handling pure Unicode have to deal with the problems. Especially with the advent of the Web, where text on a single machine can arise from many sources, this causes a significant problem.

Newline characters are used to explicitly indicate line boundaries. For more information, see Unicode Standard Annex #14, "Unicode Line Breaking Algorithm." Newlines are also handled specially in the context of regular expressions. For information, see Unicode Technical Standard #18, "Unicode Regular Expressions." For the use of these characters in markup languages, see the W3C specification, "Unicode in XML and Other Markup Languages."

Definitions

Table 5-1 provides hexadecimal values for the acronyms used in these guidelines. The acronyms shown in *Table 5-1* correspond to characters or sequences of characters. The name column shows the usual names used to refer to the characters in question, whereas the other columns show the Unicode, ASCII, and EBCDIC encoded values for the characters.

Table 5-1. Hex Values for Acronyms

Acronym	Name	Unicode	ASCII	EBCDIC	
				Default	z/OS
CR	carriage return	000D	0D	0D	0D
LF	line feed	000A	0A	25	15
CRLF	carriage return and line feed	<000D 000A>	<0D 0A>	<0D 25>	<0D 15>
NEL	next line	0085	85	15	25
VT	vertical tab	000B	0B	0B	0B
FF	form feed	000C	0C	0C	0C
LS	line separator	2028	n/a	n/a	n/a
PS	paragraph separator	2029	n/a	n/a	n/a

Encoding. Except for LS and PS, the newline characters discussed here are encoded as control codes. Many control codes were originally designed for device control but, together with TAB, the newline characters are commonly used as part of plain text. For more information on how Unicode encodes control codes, see *Section 23.1, Control Codes*.

Notation. This discussion of newline guidelines uses lowercase when referring to functions having to do with line determination, but uses the acronyms when referring to the actual characters involved. Keys on keyboards are indicated in all caps. For example:

> The line separator may be expressed by LS in Unicode text or CR on some platforms. It may be entered into text with the SHIFT-RETURN key.

EBCDIC. *Table 5-1* shows the two mappings of LF and NEL used by EBCDIC systems. The first EBCDIC column shows the default control code mapping of these characters, which is used in most EBCDIC environments. The second column shows the z/OS Unix System Services mapping of LF and NEL. That mapping arises from the use of the LF character for the newline function in C programs and in Unix environments, while text files on z/OS traditionally use NEL for the newline function.

NEL (next line) is not actually defined in 7-bit ASCII. It is defined in the ISO control function standard, ISO 6429, as a C1 control function. However, the 0x85 mapping shown in the ASCII column in *Table 5-1* is the usual way that this C1 control function is mapped in ASCII-based character encodings.

Newline Function. The acronym *NLF* (*newline function*) stands for the generic control function for indication of a new line break. It may be represented by different characters, depending on the platform, as shown in *Table 5-2*.

Table 5-2. NLF Platform Correlations

Platform	NLF Value
MacOS 9.x and earlier	CR
MacOS X	LF
Unix	LF
Windows	CRLF
EBCDIC-based OS	NEL

Line Separator and Paragraph Separator

A paragraph separator—independent of how it is encoded—is used to indicate a separation between paragraphs. A line separator indicates where a line break alone should occur, typically within a paragraph. For example:

> This is a paragraph with a line separator at this point,
> causing the word "causing" to appear on a different line, but not causing
> the typical paragraph indentation, sentence breaking, line spacing, or
> change in flush (right, center, or left paragraphs).

For comparison, line separators basically correspond to HTML <BR>, and paragraph separators to older usage of HTML <P> (modern HTML delimits paragraphs by enclosing them in <P>...</P>). In word processors, paragraph separators are usually entered using a keyboard RETURN or ENTER; line separators are usually entered using a modified RETURN or ENTER, such as SHIFT-ENTER.

A record separator is used to separate records. For example, when exchanging tabular data, a common format is to tab-separate the cells and use a CRLF at the end of a line of cells. This function is not precisely the same as line separation, but the same characters are often used.

Traditionally, *NLF* started out as a line separator (and sometimes record separator). It is still used as a line separator in simple text editors such as program editors. As platforms and programs started to handle word processing with automatic line-wrap, these characters were reinterpreted to stand for paragraph separators. For example, even such simple programs as the Windows Notepad program and the Mac SimpleText program interpret their platform's *NLF* as a paragraph separator, not a line separator.

Once *NLF* was reinterpreted to stand for a paragraph separator, in some cases another control character was pressed into service as a line separator. For example, vertical tabulation VT is used in Microsoft Word. However, the choice of character for line separator is even less standardized than the choice of character for *NLF*.

Many Internet protocols and a lot of existing text treat *NLF* as a line separator, so an implementer cannot simply treat *NLF* as a paragraph separator in all circumstances.

Recommendations

The Unicode Standard defines two unambiguous separator characters: U+2029 PARAGRAPH SEPARATOR (PS) and U+2028 LINE SEPARATOR (LS). In Unicode text, the PS and LS characters should be used wherever the desired function is unambiguous. Otherwise, the following recommendations specify how to cope with an *NLF* when converting from other character sets to Unicode, when interpreting characters in text, and when converting from Unicode to other character sets.

Note that even if an implementer knows which characters represent *NLF* on a particular platform, CR, LF, CRLF, and NEL should be treated the same on input and in interpretation. Only on output is it necessary to distinguish between them.

Converting from Other Character Code Sets

  R1   If the exact usage of any NLF is known, convert it to LS or PS.

 R1a   If the exact usage of any NLF is unknown, remap it to the platform NLF.

Recommendation R1a does not really help in interpreting Unicode text unless the implementer is the *only* source of that text, because another implementer may have left in LF, CR, CRLF, or NEL.

Interpreting Characters in Text

  R2   Always interpret PS as paragraph separator and LS as line separator.

 R2a   In word processing, interpret any NLF the same as PS.

 R2b   In simple text editors, interpret any NLF the same as LS.

In line breaking, both PS and LS terminate a line; therefore, the Unicode Line Breaking Algorithm in Unicode Standard Annex #14, "Unicode Line Breaking Algorithm," is defined such that any NLF causes a line break.

R2c   *In parsing, choose the safest interpretation.*

For example, in recommendation R2c an implementer dealing with sentence break heuristics would reason in the following way that it is safer to interpret any *NLF* as LS:

- Suppose an *NLF* were interpreted as LS, when it was meant to be PS. Because most paragraphs are terminated with punctuation anyway, this would cause misidentification of sentence boundaries in only a few cases.

- Suppose an *NLF* were interpreted as PS, when it was meant to be LS. In this case, line breaks would cause sentence breaks, which would result in significant problems with the sentence break heuristics.

Converting to Other Character Code Sets

R3   *If the intended target is known, map NLF, LS, and PS depending on the target conventions.*

For example, when mapping to Microsoft Word's internal conventions for documents, LS would be mapped to VT, and PS and any *NLF* would be mapped to CRLF.

R3a   *If the intended target is unknown, map NLF, LS, and PS to the platform newline convention (CR, LF, CRLF, or NEL).*

In Java, for example, this is done by mapping to a string `nlf`, defined as follows:

```
String nlf = System.getProperties("line.separator");
```

Input and Output

R4   *A* `readline` *function should stop at NLF, LS, FF, or PS. In the typical implementation, it does not include the NLF, LS, PS, or FF that caused it to stop.*

Because the separator is lost, the use of such a `readline` function is limited to text processing, where there is no difference among the types of separators.

R4a   *A* `writeline` *(or* `newline`*) function should convert NLF, LS, and PS according to the recommendations R3 and R3a.*

In C, `gets` is defined to terminate at a newline and replaces the newline with `'\0'`, while `fgets` is defined to terminate at a newline and includes the newline in the array into which it copies the data. C implementations interpret `'\n'` either as LF or as the underlying platform newline *NLF,* depending on where it occurs. EBCDIC C compilers substitute the relevant codes, based on the EBCDIC execution set.

Page Separator

FF is commonly used as a page separator, and it should be interpreted that way in text. When displaying on the screen, it causes the text after the separator to be forced to the next page. It is interpreted in the same way as the LS for line breaking, in parsing, or in input

segmentation such as `readline`. FF does not interrupt a paragraph, as paragraphs can and do span page boundaries.

5.9 Regular Expressions

Byte-oriented regular expression engines require extensions to handle Unicode success-fully. The following issues are involved in such extensions:

- Unicode is a large character set—regular expression engines that are adapted to handle only small character sets may not scale well.

- Unicode encompasses a wide variety of languages that can have very different characteristics than English or other Western European text.

For detailed information on the requirements of Unicode regular expressions, see Unicode Technical Standard #18, "Unicode Regular Expressions."

5.10 Language Information in Plain Text

Requirements for Language Tagging

The requirement for language information embedded in plain text data is often overstated. Many commonplace operations such as collation seldom require this extra information. In collation, for example, foreign language text is generally collated as if it were *not* in a foreign language. (See Unicode Technical Standard #10, "Unicode Collation Algorithm," for more information.) For example, an index in an English book would not sort the Slovak word "chlieb" after "czar," where it would be collated in Slovak, nor would an English atlas put the Swedish city of Örebro after Zanzibar, where it would appear in Swedish.

Text to speech is also an area where the case for embedded language information is overstated. Although language information may be useful in performing text-to-speech operations, modern software for doing acceptable text-to-speech must be so sophisticated in performing grammatical analysis of text that the extra work in determining the language is not significant in practice.

Language information can be useful in certain operations, such as spell-checking or hyphenating a mixed-language document. It is also useful in choosing the default font for a run of unstyled text; for example, the ellipsis character may have a very different appearance in Japanese fonts than in European fonts. Modern font and layout technologies produce different results based on language information. For example, the angle of the acute accent may be different for French and Polish.

Language Tags and Han Unification

A common misunderstanding about Unicode Han unification is the mistaken belief that Han characters cannot be rendered properly without language information. This idea might lead an implementer to conclude that language information must always be added to plain text using the tags. However, this implication is incorrect. The goal and methods of Han unification were to ensure that the text remained legible. Although font, size, width, and other format specifications need to be added to produce precisely the same appearance on the source and target machines, plain text remains legible in the absence of these specifications.

There should never be any confusion in Unicode, because the distinctions between the unified characters are all within the range of stylistic variations that exist in each country. No unification in Unicode should make it impossible for a reader to identify a character if it appears in a different font. Where precise font information is important, it is best conveyed in a rich text format.

Typical Scenarios. The following e-mail scenarios illustrate that the need for language information with Han characters is often overstated:

- Scenario 1. A Japanese user sends out untagged Japanese text. Readers are Japanese (with Japanese fonts). Readers see no differences from what they expect.

- Scenario 2. A Japanese user sends out an untagged mixture of Japanese and Chinese text. Readers are Japanese (with Japanese fonts) and Chinese (with Chinese fonts). Readers see the mixed text with only one font, but the text is still legible. Readers recognize the difference between the languages by the content.

- Scenario 3. A Japanese user sends out a mixture of Japanese and Chinese text. Text is marked with font, size, width, and so on, because the exact format is important. Readers have the fonts and other display support. Readers see the mixed text with different fonts for different languages. They recognize the difference between the languages by the content, and see the text with glyphs that are more typical for the particular language.

It is common even in printed matter to render passages of foreign language text in native-language fonts, just for familiarity. For example, Chinese text in a Japanese document is commonly rendered in a Japanese font.

5.11 Editing and Selection

Consistent Text Elements

As far as a user is concerned, the underlying representation of text is not a material concern, but it is important that an editing interface present a uniform implementation of what the user thinks of as characters. (See "'Characters' and Grapheme Clusters" in *Section 2.11, Combining Characters*.) The user expects them to behave as units in terms of mouse selection, arrow key movement, backspacing, and so on. For example, when such behavior is implemented, and an accented letter is represented by a sequence of base character plus a nonspacing combining mark, using the right arrow key would logically skip from the start of the base character to the end of the last nonspacing character.

In some cases, editing a user-perceived "character" or visual cluster element by element may be the preferred way. For example, a system might have the *backspace* key delete by using the underlying code point, while the *delete* key could delete an entire cluster. Moreover, because of the way keyboards and input method editors are implemented, there often may not be a one-to-one relationship between what the user thinks of as a character and the key or key sequence used to input it.

Three types of boundaries are generally useful in editing and selecting within words: cluster boundaries, stacked boundaries and atomic character boundaries.

Cluster Boundaries. Arbitrarily defined cluster boundaries may occur in scripts such as Devanagari, for which selection may be defined as applying to syllables or parts of syllables. In such cases, combining character sequences such as *ka + vowel sign a* or conjunct clusters such as *ka + halant + ta* are selected as a single unit. (See *Figure 5-3*.)

Figure 5-3. Consistent Character Boundaries

Stacked Boundaries. Stacked boundaries are generally somewhat finer than cluster boundaries. Free-standing elements (such as *vowel sign a* in Devanagari) can be independently selected, but any elements that "stack" (including vertical ligatures such as Arabic *lam + meem* in *Figure 5-3*) can be selected only as a single unit. Stacked boundaries treat default grapheme clusters as single entities, much like composite characters. (See Unicode Stan-

dard Annex #29, "Unicode Text Segmentation," for the definition of default grapheme clusters and for a discussion of how grapheme clusters can be tailored to meet the needs of defining arbitrary cluster boundaries.)

Atomic Character Boundaries. The use of atomic character boundaries is closest to selection of individual Unicode characters. However, most modern systems indicate selection with some sort of rectangular highlighting. This approach places restrictions on the consistency of editing because some sequences of characters do not linearly progress from the start of the line. When characters stack, two mechanisms are used to visually indicate partial selection: linear and nonlinear boundaries.

Linear Boundaries. Use of linear boundaries treats the entire width of the resultant glyph as belonging to the first character of the sequence, and the remaining characters in the backing-store representation as having no width and being visually afterward.

This option is the simplest mechanism. The advantage of this system is that it requires very little additional implementation work. The disadvantage is that it is never easy to select narrow characters, let alone a zero-width character. Mechanically, it requires the user to select just to the right of the nonspacing mark and drag just to the left. It also does not allow the selection of individual nonspacing marks if more than one is present.

Nonlinear Boundaries. Use of nonlinear boundaries divides any stacked element into parts. For example, picking a point halfway across a *lam + meem* ligature can represent the division between the characters. One can either allow highlighting with multiple rectangles or use another method such as coloring the individual characters.

With more work, a precomposed character can behave in deletion as if it were a composed character sequence with atomic character boundaries. This procedure involves deriving the character's decomposition on the fly to get the components to be used in simulation. For example, deletion occurs by decomposing, removing the last character, then recomposing (if more than one character remains). However, this technique does not work in general editing and selection.

In most editing systems, the code point is the smallest addressable item, so the selection and assignment of properties (such as font, color, letterspacing, and so on) cannot be done on any finer basis than the code point. Thus the accent on an "e" could not be colored differently than the base in a precomposed character, although it could be colored differently if the text were stored internally in a decomposed form.

Just as there is no single notion of text element, so there is no single notion of editing character boundaries. At different times, users may want different degrees of granularity in the editing process. Two methods suggest themselves. First, the user may set a global preference for the character boundaries. Second, the user may have alternative command mechanisms, such as Shift-Delete, which give more (or less) fine control than the default mode.

5.12 Strategies for Handling Nonspacing Marks

By following these guidelines, a programmer should be able to implement systems and routines that provide for the effective and efficient use of nonspacing marks in a wide variety of applications and systems. The programmer also has the choice between minimal techniques that apply to the vast majority of existing systems and more sophisticated techniques that apply to more demanding situations, such as higher-end desktop publishing.

In this section and the following section, the terms *nonspacing mark* and *combining character* are used interchangeably. The terms *diacritic, accent, stress mark, Hebrew point, Arabic vowel*, and others are sometimes used instead of *nonspacing mark*. (They refer to particular types of nonspacing marks.) Properly speaking, a nonspacing mark is any combining character that does not add space along the writing direction. For a formal definition of nonspacing mark, see *Section 3.6, Combination*.

A relatively small number of implementation features are needed to support nonspacing marks. Different levels of implementation are also possible. A minimal system yields good results and is relatively simple to implement. Most of the features required by such a system are simply modifications of existing software.

As nonspacing marks are required for a number of writing systems, such as Arabic, Hebrew, and those of South Asia, many vendors already have systems capable of dealing with these characters and can use their experience to produce general-purpose software for handling these characters in the Unicode Standard.

Rendering. Composite character sequences can be rendered effectively by means of a fairly simple mechanism. In simple character rendering, a nonspacing combining mark has a zero advance width, and a composite character sequence will have the same width as the base character.

Wherever a sequence of base character plus one or more nonspacing marks occurs, the glyphs for the nonspacing marks can be positioned relative to the base. The ligature mechanisms in the fonts can also substitute a glyph representing the combined form. In some cases the width of the base should change because of an applied accent, such as with "î". The ligature or contextual form mechanisms in the font can be used to change the width of the base in cases where this is required.

Other Processes. Correct multilingual comparison routines must already be able to compare a sequence of characters as one character, or one character as if it were a sequence. Such routines can also handle combining character sequences when supplied with the appropriate data. When searching strings, remember to check for additional nonspacing marks in the target string that may affect the interpretation of the last matching character.

Line breaking algorithms generally use state machines for determining word breaks. Such algorithms can be easily adapted to prevent separation of nonspacing marks from base characters. (See also the discussion in *Section 5.6, Normalization*. For details in particular contexts, see Unicode Technical Standard #10, "Unicode Collation Algorithm"; Unicode

Standard Annex #14, "Unicode Line Breaking Algorithm"; and Unicode Standard Annex #29, "Unicode Text Segmentation.")

Keyboard Input

A common implementation for the input of combining character sequences is the use of *dead keys*. These keys match the mechanics used by typewriters to generate such sequences through overtyping the base character after the nonspacing mark. In computer implementations, keyboards enter a special state when a dead key is pressed for the accent and emit a precomposed character only when one of a limited number of "legal" base characters is entered. It is straightforward to adapt such a system to emit combining character sequences or precomposed characters as needed.

Typists, especially in the Latin script, are trained on systems that work using dead keys. However, many scripts in the Unicode Standard (including the Latin script) may be implemented according to the handwriting sequence, in which users type the base character first, *followed* by the accents or other nonspacing marks (see *Figure 5-4*).

Figure 5-4. Dead Keys Versus Handwriting Sequence
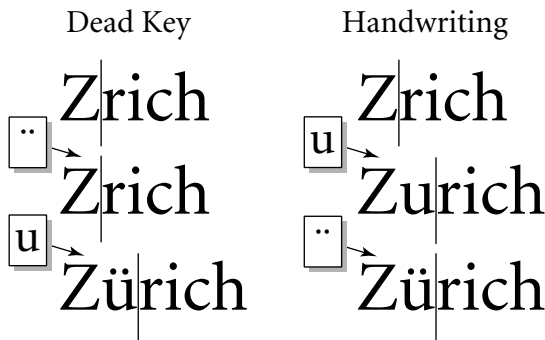
In the case of handwriting sequence, each keystroke produces a distinct, natural change on the screen; there are no hidden states. To add an accent to any existing character, the user positions the insertion point (*caret*) after the character and types the accent.

Truncation

There are two types of truncation: truncation by character count and truncation by displayed width. Truncation by character count can entail loss (be lossy) or be lossless.

Truncation by character count is used where, due to storage restrictions, a limited number of characters can be entered into a field; it is also used where text is broken into buffers for transmission and other purposes. The latter case can be lossless if buffers are recombined seamlessly before processing or if lookahead is performed for possible combining character sequences straddling buffers.

When fitting data into a field of limited storage length, some information will be lost. The preferred position for truncating text in that situation is on a grapheme cluster boundary. As *Figure 5-5* shows, such truncation can mean truncating at an earlier point than the last character that would have fit within the physical storage limitation. (See Unicode Standard Annex #29, "Unicode Text Segmentation.")

Figure 5-5. Truncating Grapheme Clusters
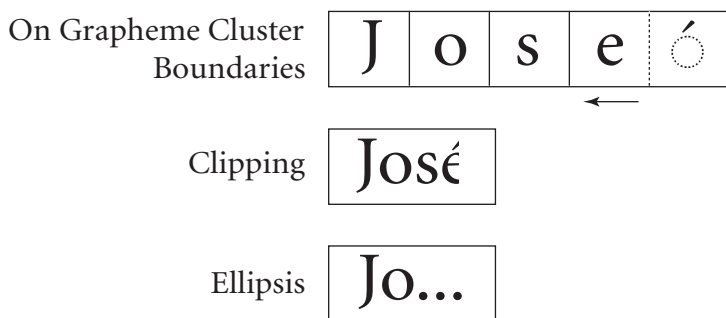
Truncation by displayed width is used for visual display in a narrow field. In this case, truncation occurs on the basis of the width of the resulting string rather than on the basis of a character count. In simple systems, it is easiest to truncate by width, starting from the end and working backward by subtracting character widths as one goes. Because a trailing nonspacing mark does not contribute to the measurement of the string, the result will not separate nonspacing marks from their base characters.

If the textual environment is more sophisticated, the widths of characters may depend on their context, due to effects such as kerning, ligatures, or contextual formation. For such systems, the width of a precomposed character, such as an "ï", may be different than the width of a narrow base character alone. To handle these cases, a final check should be made on any truncation result derived from successive subtractions.

A different option is simply to clip the characters graphically. Unfortunately, this may result in clipping off part of a character, which can be visually confusing. Also, if the clipping occurs between characters, it may not give any visual feedback that characters are being omitted. A graphic or ellipsis can be used to give this visual feedback.

5.13 Rendering Nonspacing Marks

This discussion assumes the use of proportional fonts, where the widths of individual characters can vary. Various techniques can be used with monospaced fonts. In general, however, it is possible to get only a semblance of a correct rendering for most scripts in such fonts.

When rendering a sequence consisting of more than one nonspacing mark, the nonspacing marks should, by default, be stacked outward from the base character. That is, if two nonspacing marks appear over a base character, then the first nonspacing mark should appear on top of the base character, and the second nonspacing mark should appear on top of the first. If two nonspacing marks appear under a base character, then the first nonspacing mark should appear beneath the base character, and the second nonspacing mark should appear below the first (see *Section 2.11, Combining Characters*). This default treatment of multiple, potentially interacting nonspacing marks is known as the inside-out rule (see *Figure 5-6*).

Figure 5-6. Inside-Out Rule
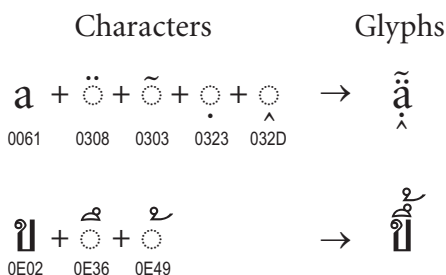
This default behavior may be altered based on typographic preferences or on knowledge of the specific orthographic treatment to be given to multiple nonspacing marks in the context of a particular writing system. For example, in the modern Vietnamese writing system, an acute or grave accent (serving as a tone mark) may be positioned slightly to one side of a circumflex accent rather than directly above it. If the text to be displayed is known to employ a different typographic convention (either implicitly through knowledge of the language of the text or explicitly through rich text-style bindings), then an alternative positioning may be given to multiple nonspacing marks instead of that specified by the default inside-out rule.

Fallback Rendering. Several methods are available to deal with an unknown composed character sequence that is outside of a fixed, renderable set (see *Figure 5-7*). One method (*Show Hidden*) indicates the inability to draw the sequence by drawing the base character first and then rendering the nonspacing mark as an individual unit, with the nonspacing mark positioned on a dotted circle. (This convention is used in the Unicode code charts.)

Another method (*Simple Overlap*) uses a default fixed position for an overlapping zero-width nonspacing mark. This position is generally high enough to make sure that the mark

Figure 5-7. Fallback Rendering

Ĝĝ G͜ĝ͜ Ĝ̣ĝ̣

"Ideal"   "Show   "Simple
         Hidden"   Overlap"

does not collide with capital letters. This will mean that this mark is placed too high above many lowercase letters. For example, the default positioning of a circumflex can be above the ascent, which will place it above capital letters. Even though the result will not be particularly attractive for letters such as *g-circumflex*, the result should generally be recognizable in the case of single nonspacing marks.

In a degenerate case, a nonspacing mark occurs as the first character in the text or is separated from its base character by a *line separator*, *paragraph separator,* or other format character that causes a positional separation. This result is called a defective combining character sequence (see *Section 3.6, Combination*). Defective combining character sequences should be rendered as if they had a *no-break space* as a base character. (See *Section 7.9, Combining Marks.*)

Bidirectional Positioning. In bidirectional text, the nonspacing marks are reordered *with* their base characters; that is, they visually apply to the same base character after the algorithm is used (see *Figure 5-8*). There are a few ways to accomplish this positioning.

The simplest method is similar to the *Simple Overlap* fallback method. In the Bidirectional Algorithm, combining marks take the level of their base character. In that case, Arabic and Hebrew nonspacing marks would come to the left of their base characters. The font is designed so that instead of overlapping to the left, the Arabic and Hebrew nonspacing marks overlap to the right. In *Figure 5-8*, the "glyph metrics" line shows the pen start and end for each glyph with such a design. After aligning the start and end points, the final result shows each nonspacing mark attached to the corresponding base letter. More sophisticated rendering could then apply the positioning methods outlined in the next section.

Some rendering software may require keeping the nonspacing mark glyphs consistently ordered to the right of the base character glyphs. In that case, a second pass can be done after producing the "screen order" to put the odd-level nonspacing marks on the right of their base characters. As the levels of nonspacing marks will be the same as their base characters, this pass can swap the order of nonspacing mark glyphs and base character glyphs in right-to-left (odd) levels. (See Unicode Standard Annex #9, "Unicode Bidirectional Algorithm.")

Figure 5-8. Bidirectional Placement

Backing Store

Screen Order

Glyph Metrics

Aligned Glyphs
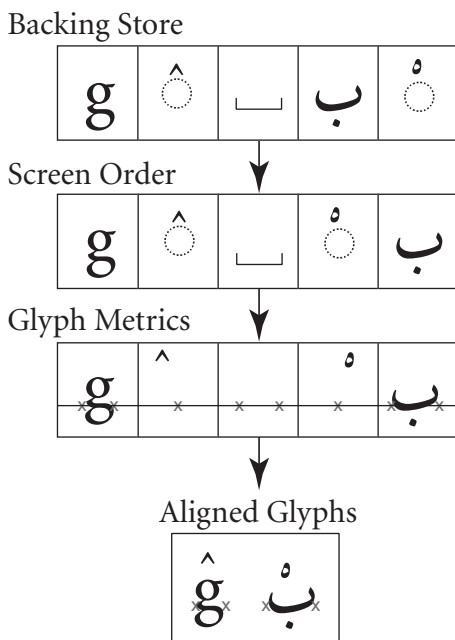
Justification. Typically, full justification of text adds extra space at space characters so as to widen a line; however, if there are too few (or no) space characters, some systems add extra letterspacing between characters (see *Figure 5-9*). This process needs to be modified if zero-width nonspacing marks are present in the text. Otherwise, if extra justifying space is added after the base character, it can have the effect of visually separating the nonspacing mark from its base.

Figure 5-9. Justification

Zürich

Z u ¨ r i c h   66 points/6 positions
                  = 11 points per position

Z ü r i c h   66 points/5 positions
                  = 13.2 points per position
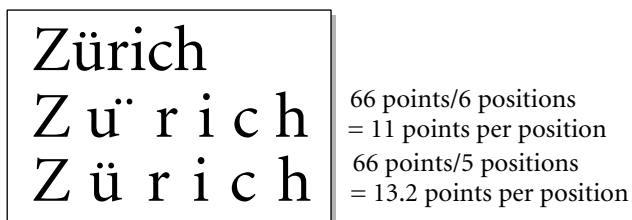
Because nonspacing marks always follow their base character, proper justification adds letterspacing between characters only if the second character is a base character.

Canonical Equivalence

Canonical equivalence must be taken into account in rendering multiple accents, so that any two canonically equivalent sequences display as the same. This is particularly important when the canonical order is not the customary keyboarding order, which happens in Arabic with vowel signs or in Hebrew with points. In those cases, a rendering system may be presented with either the typical typing order or the canonical order resulting from normalization, as shown in *Table 5-3*.

Table 5-3. Typing Order Differing from Canonical Order

Typical Typing Order	Canonical Order
U+0631 ر ARABIC LETTER REH + U+0651 ◌ّ ARABIC SHADDA + U+064B ◌ً ARABIC FATHATAN	U+0631 ر ARABIC LETTER REH + U+064B ◌ً ARABIC FATHATAN + U+0651 ◌ّ ARABIC SHADDA

With a restricted repertoire of nonspacing mark sequences, such as those required for Arabic, a ligature mechanism can be used to get the right appearance, as described earlier. When a fallback mechanism for placing accents based on their combining class is employed, the system should logically reorder the marks before applying the mechanism.

Rendering systems should handle *any* of the canonically equivalent orders of combining marks. This is not a performance issue: the amount of time necessary to reorder combining marks is insignificant compared to the time necessary to carry out other work required for rendering.

A rendering system can reorder the marks internally if necessary, as long as the resulting sequence is canonically equivalent. In particular, any permutation of the non-zero combining class values can be used for a canonical-equivalent internal ordering. For example, a rendering system could internally permute weights to have U+0651 ARABIC SHADDA precede all vowel signs. This would use the remapping shown in *Table 5-4*.

Table 5-4. Permuting Combining Class Weights

Combining Class		Internal Weight
27	→	33
28	→	27
29	→	28
30	→	29
31	→	30
32	→	31
33	→	32

Only non-zero combining class values can be changed, and they can be permuted *only*, not combined or split. This can be restated as follows:

- Two characters that have the same combining class values cannot be given distinct internal weights.

- Two characters that have distinct combining class values cannot be given the same internal weight.

- Characters with a combining class of zero must be given an internal weight of zero.

Positioning Methods

A number of methods are available to position nonspacing marks so that they are in the correct location relative to the base character and previous nonspacing marks.

Positioning with Ligatures. A fixed set of combining character sequences can be rendered effectively by means of fairly simple substitution, as shown in *Figure 5-10*.

Figure 5-10. Positioning with Ligatures

$$a \; + \; \ddot{\circ} \; \to \; ä$$
$$A \; + \; \ddot{\circ} \; \to \; Ä$$
$$f \; + \; i \; \to \; fi$$

Wherever the glyphs representing a sequence of <base character, nonspacing mark> occur, a glyph representing the combined form is substituted. Because the nonspacing mark has a zero advance width, the composed character sequence will automatically have the same width as the base character. More sophisticated text rendering systems may take additional measures to account for those cases where the composed character sequence kerns differently or has a slightly different advance width than the base character.

Positioning with ligatures is perhaps the simplest method of supporting nonspacing marks. Whenever there is a small, fixed set, such as those corresponding to the precomposed characters of ISO/IEC 8859-1 (Latin-1), this method is straightforward to apply. Because the composed character sequence almost always has the same width as the base character, rendering, measurement, and editing of these characters are much easier than for the general case of ligatures.

If a combining character sequence does not form a ligature, then either positioning with contextual forms or positioning with enhanced kerning can be applied. If they are not available, then a fallback method can be used.

Positioning with Contextual Forms. A more general method of dealing with positioning of nonspacing marks is to use contextual formation (see *Figure 5-11*). In this case for Devanagari, a consonant RA is rendered with a nonspacing glyph (*reph*) positioned above a base consonant. (See "Rendering Devanagari" in *Section 12.1, Devanagari.*) Depending on the position of the stem for the corresponding base consonant glyph, a contextual choice is

made between *reph* glyphs with different side bearings, so that the tip of the *reph* will be placed correctly with respect to the base consonant's stem. Base glyphs generally fall into a fairly small number of classes, depending on their general shape and width, so a corresponding number of contextually distinct glyphs for the nonspacing mark suffice to produce correct rendering.

Figure 5-11. Positioning with Contextual Forms
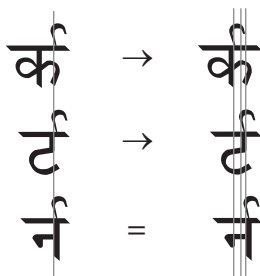
In general cases, a number of different heights of glyphs can be chosen to allow stacking of glyphs, at least for a few deep. (When these bounds are exceeded, then the fallback methods can be used.) This method can be combined with the ligature method so that in specific cases ligatures can be used to produce fine variations in position and shape.

Positioning with Enhanced Kerning. A third technique for positioning diacritics is an extension of the normal process of kerning to be both horizontal and vertical (see *Figure 5-12*). Typically, kerning maps from pairs of glyphs to a positioning offset. For example, in the word "To" the "o" should nest slightly under the "T". An extension of this system maps to both a *vertical* and a *horizontal* offset, allowing glyphs to be positioned arbitrarily.

Figure 5-12. Positioning with Enhanced Kerning
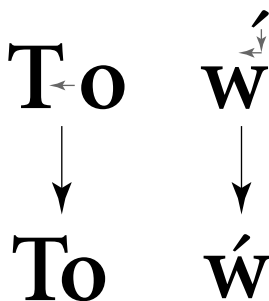
For effective use in the general case, the kerning process must be extended to handle more than simple kerning pairs, as multiple diacritics may occur after a base letter.

Positioning with enhanced kerning can be combined with the ligature method so that in specific cases ligatures can be used to produce fine variations in position and shape.

5.14 Locating Text Element Boundaries

A string of Unicode-encoded text often needs to be broken up into text elements program-matically. Common examples of text elements include what users think of as characters, words, lines, and sentences. The precise determination of text elements may vary according to locale, even as to what constitutes a "character." The goal of matching user perceptions cannot always be met, because the text alone does not always contain enough information to decide boundaries unambiguously. For example, the *period* (U+002E FULL STOP) is used ambiguously—sometimes for end-of-sentence purposes, sometimes for abbreviations, and sometimes for numbers. In most cases, however, programmatic text boundaries can match user perceptions quite closely, or at least not surprise the user.

Rather than concentrate on algorithmically searching for text elements themselves, a simpler computation looks instead at detecting the *boundaries* between those text elements. Precise definitions of the default Unicode mechanisms for determining such text element boundaries are found in Unicode Standard Annex #14, "Unicode Line Breaking Algorithm," and in Unicode Standard Annex #29, "Unicode Text Segmentation."

5.15 Identifiers

A common task facing an implementer of the Unicode Standard is the provision of a parsing and/or lexing engine for identifiers. To assist in the standard treatment of identifiers in Unicode character-based parsers, a set of guidelines is provided in Unicode Standard Annex #31, "Unicode Identifier and Pattern Syntax," as a recommended default for the definition of identifier syntax. That document provides details regarding the syntax and conformance considerations. Associated data files defining the character properties referred to by the identifier syntax can be found in the Unicode Character Database.

5.16 Sorting and Searching

Sorting and searching overlap in that both implement degrees of *equivalence* of terms to be compared. In the case of searching, equivalence defines when terms match (for example, it determines when case distinctions are meaningful). In the case of sorting, equivalence affects the proximity of terms in a sorted list. These determinations of equivalence often depend on the application and language, but for an implementation supporting the Unicode Standard, sorting and searching must always take into account the Unicode character equivalence and canonical ordering defined in *Chapter 3, Conformance.*

Culturally Expected Sorting and Searching

Sort orders vary from culture to culture, and many specific applications require variations. Sort order can be by word or sentence, case-sensitive or case-insensitive, ignoring accents or not. It can also be either phonetic or based on the appearance of the character, such as ordering by stroke and radical for East Asian ideographs. Phonetic sorting of Han characters requires use of either a lookup dictionary of words or special programs to maintain an associated phonetic spelling for the words in the text.

Languages vary not only regarding which types of sorts to use (and in which order they are to be applied), but also in what constitutes a fundamental element for sorting. For example, Swedish treats U+00C4 ʟᴀᴛɪɴ ᴄᴀᴘɪᴛᴀʟ ʟᴇᴛᴛᴇʀ ᴀ ᴡɪᴛʜ ᴅɪᴀᴇʀᴇsɪs as an individual letter, sorting it after *z* in the alphabet; German, however, sorts it either like *ae* or like other accented forms of *ä* following *a*. Spanish traditionally sorted the digraph *ll* as if it were a letter between *l* and *m*. Examples from other languages (and scripts) abound.

As a result, it is not possible either to arrange characters in an encoding such that simple binary string comparison produces the desired collation order or to provide single-level sort-weight tables. The latter implies that character encoding details have only an indirect influence on culturally expected sorting.

Unicode Technical Standard #10, "Unicode Collation Algorithm" (UCA), describes the issues involved in culturally appropriate sorting and searching, and provides a specification for how to compare two Unicode strings while remaining conformant to the requirements of the Unicode Standard. The UCA also supplies the Default Unicode Collation Element Table as the data specifying the default collation order. Searching algorithms, whether brute-force or sublinear, can be adapted to provide language-sensitive searching as described in the UCA.

Language-Insensitive Sorting

In some circumstances, an application may need to do language-insensitive sorting—that is, sorting of textual data without consideration of language-specific cultural expectations about how strings should be ordered. For example, a temporary index may need only to be in *some* well-defined order, but the exact details of the order may not matter or be visible to users. However, even in these circumstances, implementers should be aware of some issues.

First, some subtle differences arise in binary ordering between the three Unicode encoding forms. Implementations that need to do only binary comparisons between Unicode strings still need to take this issue into account so as not to create interoperability problems between applications using different encoding forms. See *Section 5.17, Binary Order*, for further discussion.

Many applications of sorting or searching need to be case-insensitive, even while not caring about language-specific differences in ordering. This is the result of the design of protocols that may be very old but that are still of great current relevance. Traditionally, implementations did case-insensitive comparison by effectively mapping both strings to uppercase before doing a binary comparison. This approach is, however, not more generally extensible to the full repertoire of the Unicode Standard. The correct approach to case-insensitive comparison is to make use of case folding, as described in *Section 5.18, Case Mappings*.

Searching

Searching is subject to many of the same issues as comparison. Other features are often added, such as only matching words (that is, where a word boundary appears on each side of the match). One technique is to code a fast search for a weak match. When a candidate is found, additional tests can be made for other criteria (such as matching diacritics, word match, case match, and so on).

When searching strings, it is necessary to check for trailing nonspacing marks in the target string that may affect the interpretation of the last matching character. That is, a search for "San Jose" may find a match in the string "Visiting San José, Costa Rica, is a...". If an exact (diacritic) match is desired, then this match should be rejected. If a weak match is sought, then the match should be accepted, but any trailing nonspacing marks should be included when returning the location and length of the target substring. The mechanisms discussed in Unicode Standard Annex #29, "Unicode Text Segmentation," can be used for this purpose.

One important application of weak equivalence is case-insensitive searching. Many traditional implementations map both the search string and the target text to uppercase. However, case mappings are language-dependent and *not* unambiguous. The preferred method of implementing case insensitivity is described in *Section 5.18, Case Mappings*.

A related issue can arise because of inaccurate mappings from external character sets. To deal with this problem, characters that are easily confused by users can be kept in a weak equivalency class (đ *d-bar,* ð *eth,* Đ *capital d-bar,* Ð *capital eth*). This approach tends to do a better job of meeting users' expectations when searching for named files or other objects.

Sublinear Searching

International searching is clearly possible using the information in the collation, just by using brute force. However, this tactic requires an $O(m*n)$ algorithm in the worst case and an $O(m)$ algorithm in common cases, where n is the number of characters in the pattern that is being searched for and m is the number of characters in the target to be searched.

A number of algorithms allow for fast searching of simple text, using sublinear algorithms. These algorithms have only $O(m/n)$ complexity in common cases by skipping over characters in the target. Several implementers have adapted one of these algorithms to search text pre-transformed according to a collation algorithm, which allows for fast searching with native-language matching (see *Figure 5-13*).

Figure 5-13. Sublinear Searching

```
T h e _ q u i c k _ b r o w n …
q u i c k̸
  q̶ ̶u̶ ̶i̶ ̶c̶ ̶k̶
    q̶ ̶u̶ ̶i̶ ̶c̶ ̶k̶
      q̶ ̶u̶ ̶i̶ ̶c̶ ̶k̶
        q u i c c⃝k⃝
```

The main problems with adapting a language-aware collation algorithm for sublinear searching relate to multiple mappings and ignorables. Additionally, sublinear algorithms precompute tables of information. Mechanisms like the two-stage tables shown in *Figure 5-1* are efficient tools in reducing memory requirements.

5.17 Binary Order

When comparing text that is visible to end users, a correct linguistic sort should be used, as described in *Section 5.16, Sorting and Searching.* However, in many circumstances the only requirement is for a fast, well-defined ordering. In such cases, a binary ordering can be used.

Not all encoding forms of Unicode have the same binary order. UTF-8 and UTF-32 data, and UTF-16 data containing only BMP characters, sort in code point order, whereas UTF-16 data containing a mix of BMP and supplementary characters does not. This is because supplementary characters are encoded in UTF-16 with pairs of surrogate code units that have lower values ($D800_{16}..DFFF_{16}$) than some BMP code points.

Furthermore, when UTF-16 or UTF-32 data is serialized using one of the Unicode encoding schemes and compared byte-by-byte, the resulting byte sequences may or may not have the same binary ordering, because swapping the order of bytes will affect the overall ordering of the data. Due to these factors, text in the UTF-16BE, UTF-16LE, and UTF-32LE encoding schemes does not sort in code point order.

In general, the default binary sorting order for Unicode text should be code point order. However, it may be necessary to match the code unit ordering of a particular encoding form (or the byte ordering of a particular encoding scheme) so as to duplicate the ordering used in a different application.

Some sample routines are provided here for sorting one encoding form in the binary order of another encoding form.

UTF-8 in UTF-16 Order

The following comparison function for UTF-8 yields the same results as UTF-16 binary comparison. In the code, notice that it is necessary to do extra work only once per string, not once per byte. That work can consist of simply remapping through a small array; there are no extra conditional branches that could slow down the processing.

```
int strcmp8like16(unsigned char* a, unsigned char* b) {
  while (true) {
    int ac = *a++;
    int bc = *b++;
    if (ac != bc) return rotate[ac] - rotate[bc];
    if (ac == 0) return 0;
  }
}
```

```
static char rotate[256] =
  {0x00, ...,                                                    0x0F,
   0x10, ...,                                                    0x1F,

      .                                                             .
      .                                                             .
      .                                                             .

   0xD0, ...,                                                    0xDF,
   0xE0, ...,                                    0xED, 0xF3, 0xF4,
   0xEE, 0xEF, 0xF0, 0xF1, 0xF2, 0xF5, ...,                     0xFF};
```

The rotate array is formed by taking an array of 256 bytes from 0x00 to 0xFF, and rotating 0xEE to 0xF4, the initial byte values of UTF-8 for the code points in the range U+E000..U+10FFFF. These rotated values are shown in boldface. When this rotation is performed on the initial bytes of UTF-8, it has the effect of making code points U+10000..U+10FFFF sort below U+E000..U+FFFF, thus mimicking the ordering of UTF-16.

UTF-16 in UTF-8 Order

The following code can be used to sort UTF-16 in code point order. As in the routine for sorting UTF-8 in UTF-16 order, the extra cost is incurred once per function call, not once per character.

```
int strcmp16like8(Unichar* a, Unichar* b) {
  while (true) {
    int ac = *a++;
    int bc = *b++;
    if (ac != bc) {
      return (Unichar)(ac + utf16Fixup[ac>>11]) -
             (Unichar)(bc + utf16Fixup[bc>>11]);
    }
    if (ac == 0) return 0;
  }
}
static const Unichar utf16Fixup[32]={
  0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0x2000, 0xf800, 0xf800, 0xf800, 0xf800
};
```

This code uses `Unichar` as an unsigned 16-bit integral type. The construction of the `utf16Fixup` array is based on the following concept. The range of UTF-16 values is

divided up into thirty-two 2K chunks. The 28th chunk corresponds to the values 0xD800..0xDFFF—that is, the surrogate code units. The 29th through 32nd chunks correspond to the values 0xE000..0xFFFF. The addition of 0x2000 to the surrogate code units rotates them up to the range 0xF800..0xFFFF. Adding 0xF800 to the values 0xE000..0xFFFF and ignoring the unsigned integer overflow rotates them down to the range 0xD800..0xF7FF. Calculating the final difference for the return from the rotated values produces the same result as basing the comparison on code points, rather than the UTF-16 code units. The use of the hack of unsigned integer overflow on addition avoids the need for a conditional test to accomplish the rotation of values.

Note that this mechanism works correctly only on well-formed UTF-16 text. A modified algorithm must be used to operate on 16-bit Unicode strings that could contain isolated surrogates.

5.18 Case Mappings

Case is a normative property of characters in specific alphabets such as Latin, Greek, Cyrillic, Armenian, and archaic Georgian, whereby characters are considered to be variants of a single letter. These variants, which may differ markedly in shape and size, are called the uppercase letter (also known as capital or majuscule) and the lowercase letter (also known as small or minuscule). The uppercase letter is generally larger than the lowercase letter. Alphabets with case differences are called *bicameral*; those without are called *unicameral*. For example, the archaic Georgian script contained upper- and lowercase pairs, but they are not used in modern Georgian. See *Section 7.7, Georgian*, for more information.

The case mappings in the Unicode Character Database (UCD) are normative. This follows from their use in defining the case foldings in CaseFolding.txt and from the use of case foldings to define case-insensitive identifiers in Unicode Standard Annex #31, "Unicode Identifier and Pattern Syntax." However, the normative status of case mappings does not preclude the adaptation of case mapping processes to local conventions, as discussed below. See also the Unicode Common Locale Data Repository (CLDR), in *Section B.6, Other Unicode Online Resources*, for extensive data regarding local and language-specific casing conventions.

Titlecasing

Titlecasing refers to a casing practice wherein the first letter of a word is an uppercase letter and the rest of the letters are lowercase. This typically applies, for example, to initial words of sentences and to proper nouns. Depending on the language and orthographic practice, this convention may apply to other words as well, as for common nouns in German.

Titlecasing also applies to entire strings, as in instances of headings or titles of documents, for which multiple words are titlecased. The choice of which words to titlecase in headings and titles is dependent on language and local conventions. For example, "The Merry Wives of Windsor" is the appropriate titlecasing of that play's name in English, with the word "of" not titlecased. In German, however, the title is "Die lustigen Weiber von Windsor," and both "lustigen" and "von" are not titlecased. In French even fewer words are titlecased: "Les joyeuses commères de Windsor."

Moreover, the determination of what actually constitutes a word is language dependent, and this can influence which letter or letters of a "word" are uppercased when titlecasing strings. For example *l'arbre* is considered two words in French, whereas *can't* is considered one word in English.

The need for a normative Titlecase_Mapping property in the Unicode Standard derives from the fact that the standard contains certain digraph characters for compatibility. These digraph compatibility characters, such as U+01F3 "dz" ʟᴀᴛɪɴ ꜱᴍᴀʟʟ ʟᴇᴛᴛᴇʀ ᴅᴢ, require one form when being uppercased, U+01F1 "DZ" ʟᴀᴛɪɴ ᴄᴀᴘɪᴛᴀʟ ʟᴇᴛᴛᴇʀ ᴅᴢ, and another form when being titlecased, U+01F2 "Dz" ʟᴀᴛɪɴ ᴄᴀᴘɪᴛᴀʟ ʟᴇᴛᴛᴇʀ ᴅ ᴡɪᴛʜ ꜱᴍᴀʟʟ ʟᴇᴛᴛᴇʀ ᴢ. The latter form is informally referred to as a *titlecase character*, because it is mixed case, with the first letter uppercase. Most characters in the standard have identical values for

their Titlecase_Mapping and Uppercase_Mapping; however, the two values are distinguished for these few digraph compatibility characters.

Complications for Case Mapping

A number of complications to case mappings occur once the repertoire of characters is expanded beyond ASCII.

Change in Length. Case mappings may produce strings of different lengths than the original. For example, the German character U+00DF ß LATIN SMALL LETTER SHARP S expands when uppercased to the sequence of two characters "SS". Such expansion also occurs where there is no precomposed character corresponding to a case mapping, such as with U+0149 ŉ LATIN SMALL LETTER N PRECEDED BY APOSTROPHE. The maximum string expansion as a result of case mapping in the Unicode Standard is three. For example, uppercasing U+0390 ΐ GREEK SMALL LETTER IOTA WITH DIALYTIKA AND TONOS results in three characters.

The lengths of case-mapped strings may also differ from their originals depending on the Unicode encoding form. For example, the Turkish strings "topkapı" (with a *dotless i*) and "TOPKAPI" have the same number of characters and are the same length in UTF-16 and UTF-32; however, in UTF-8, the representation of the uppercase form takes only seven bytes, whereas the lowercase form takes eight bytes. By comparison, the German strings "heiß" and "HEISS" have a different number of characters and differ in length in UTF-16 and UTF-32, but in UTF-8 both strings are encoded using the same number of bytes.

***Greek* iota subscript.** The character U+0345 ͅ COMBINING GREEK YPOGEGRAMMENI (*iota subscript*) requires special handling. As discussed in *Section 7.2, Greek*, the iota-subscript characters used to represent ancient text have special case mappings. Normally, the uppercase and lowercase forms of alpha-iota-subscript will map back and forth. In some instances, uppercase words should be transformed into their older spellings by removing accents and changing the iota subscript into a capital iota (and perhaps even removing spaces).

Context-dependent Case Mappings. Characters may have different case mappings, depending on the context surrounding the character in the original string. For example, U+03A3 "Σ" GREEK CAPITAL LETTER SIGMA lowercases to U+03C3 "σ" GREEK SMALL LETTER SIGMA if it is followed by another letter, but lowercases to U+03C2 "ς" GREEK SMALL LETTER FINAL SIGMA if it is not.

Because only a few context-sensitive case mappings exist, and because they involve only a very few characters, implementations may choose to hard-code the treatment of these characters for casing operations rather than using data-driven code based on the Unicode Character Database. However, if this approach is taken, each time the implementation is upgraded to a new version of the Unicode Standard, hard-coded casing operations should be checked for consistency with the updated data. See SpecialCasing.txt in the Unicode Character Database for details of context-sensitive case mappings.

Locale-dependent Case Mappings. The principal example of a case mapping that depends on the locale is Turkish, where U+0131 "ı" LATIN SMALL LETTER DOTLESS I maps to

U+0049 "I" ʟᴀᴛɪɴ ᴄᴀᴘɪᴛᴀʟ ʟᴇᴛᴛᴇʀ ɪ and U+0069 "i" ʟᴀᴛɪɴ sᴍᴀʟʟ ʟᴇᴛᴛᴇʀ ɪ maps to U+0130 " İ " ʟᴀᴛɪɴ ᴄᴀᴘɪᴛᴀʟ ʟᴇᴛᴛᴇʀ ɪ ᴡɪᴛʜ ᴅᴏᴛ ᴀʙᴏᴠᴇ. *Figure 5-14* shows the uppercase mapping for Turkish *i* and canonically equivalent sequences.

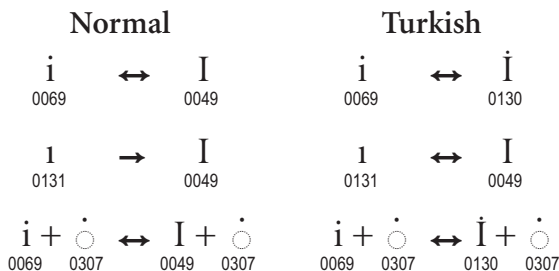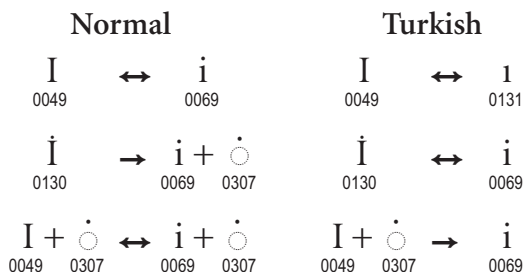**Figure 5-14.** Uppercase Mapping for Turkish I

Figure 5-15 shows the lowercase mapping for Turkish *i*.

**Figure 5-15.** Lowercase Mapping for Turkish I

In both of the Turkish case mapping figures, a mapping with a double-sided arrow round-trips—that is, the opposite case mapping results in the original sequence. A mapping with a single-sided arrow does not round-trip.

Caseless Characters. Because many characters are really caseless (most of the IPA block, for example) and have no matching uppercase, the process of uppercasing a string does *not* mean that it will no longer contain any lowercase letters.

***German* sharp s.** The German *sharp s* character has several complications in case mapping. Not only does its uppercase mapping expand in length, but its default case-pairings are asymmetrical. The default case mapping operations follow standard German orthography, which uses the string "SS" as the regular uppercase mapping for U+00DF ß ʟᴀᴛɪɴ sᴍᴀʟʟ ʟᴇᴛᴛᴇʀ sʜᴀʀᴘ s. In contrast, the alternate, single character uppercase form, U+1E9E ʟᴀᴛɪɴ ᴄᴀᴘɪᴛᴀʟ ʟᴇᴛᴛᴇʀ sʜᴀʀᴘ s, is intended for typographical representations of signage and uppercase titles, and in other environments where users require the *sharp s* to be preserved in uppercase. Overall, such usage is uncommon. Thus, when using the default Unicode casing operations, *capital sharp s* will lowercase to *small sharp s*, but not vice versa: *small sharp*

s uppercases to "SS", as shown in *Figure 5-16*. A tailored casing operation is needed in circumstances requiring *small sharp s* to uppercase to *capital sharp s*.

Figure 5-16. Casing of German Sharp S

Default Casing   Tailored Casing
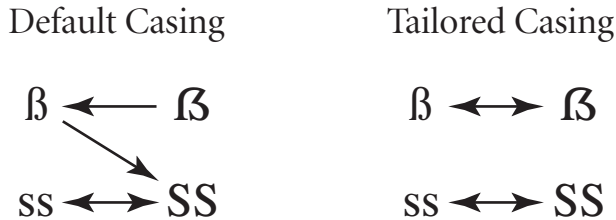
Reversibility

No casing operations are reversible. For example:

> `toUpperCase(toLowerCase(`"John Brown"`))` → "JOHN BROWN"

> `toLowerCase(toUpperCase(`"John Brown"`))` → "john brown"

There are even single words like *vederLa* in Italian or the name *McGowan* in English, which are neither upper-, lower-, nor titlecase. This format is sometimes called *inner-caps*—or more informally *camelcase*—and it is often used in programming and in Web names. Once the string "McGowan" has been uppercased, lowercased, or titlecased, the original cannot be recovered by applying another uppercase, lowercase, or titlecase operation. There are also single characters that do not have reversible mappings, such as the Greek sigmas.

For word processors that use a single command-key sequence to toggle the selection through different casings, it is recommended to save the original string and return to it via the sequence of keys. The user interface would produce the following results in response to a series of command keys. In the following example, notice that the original string is restored every fourth time.

1. The quick brown

2. THE QUICK BROWN

3. the quick brown

4. The Quick Brown

5. The quick brown (repeating from here on)

Uppercase, titlecase, and lowercase can be represented in a word processor by using a character style. Removing the character style restores the text to its original state. However, if this approach is taken, any spell-checking software needs to be aware of the case style so that it can check the spelling against the actual appearance.

Caseless Matching

Caseless matching is implemented using *case folding*, which is the process of mapping characters of different case to a single form, so that case differences in strings are erased. Case folding allows for fast caseless matches in lookups because only binary comparison is required. It is more than just conversion to lowercase. For example, it correctly handles cases such as the Greek sigma, so that "όσος" and "ΟΣΟΣ" will match.

Normally, the original source string is not replaced by the folded string because that substitution may erase important information. For example, the name "Marco di Silva" would be folded to "marco di silva," losing the information regarding which letters are capitalized. Typically, the original string is stored along with a case-folded version for fast comparisons.

The CaseFolding.txt file in the Unicode Character Database is used to perform locale-independent case folding. This file is generated from the case mappings in the Unicode Character Database, using both the single-character mappings and the multicharacter mappings. It folds all characters having different case forms together into a common form. To compare two strings for caseless matching, one can fold each string using this data and then use a binary comparison.

Case folding logically involves a set of equivalence classes constructed from the Unicode Character Database case mappings as follows.

For each character X in Unicode, apply the following rules in order:

R1   *If X is already in an equivalence class, continue to the next character. Otherwise, form a new equivalence class and add X.*

R2   *Add any other character that uppercases, lowercases, or titlecases to anything in the equivalence class.*

R3   *Add any other characters to which anything in the equivalence class uppercases, lowercases, or titlecases.*

R4   *Repeat R2 and R3 until nothing further is added.*

R5   *From each class, one representative element (a single lowercase letter where possible) is chosen to be the common form.*

For rule R5, it is preferable to choose a single lowercase letter for the common form, but this is not possible in all instances. For case folding of Cherokee letters, for example, a single *uppercase* letter must be chosen instead, because the uppercase letters for Cherokee were encoded in an earlier version of the Unicode Standard, and the lowercase letters were encoded in a later version. This choice is required to keep case folding stable across Unicode versions.

Each equivalence class is completely disjoint from all the others, and every Unicode character is in one equivalence class. CaseFolding.txt thus contains the mappings from other characters in the equivalence classes to their common forms. As an exception, the case foldings for dotless i and dotted I do not follow the derivation algorithm for all other case foldings. Instead, their case foldings are hard-coded in the derivation for best default matching

behavior. There are alternate case foldings for these characters, which can be used for case folding for Turkic languages. However, the use of those alternate case foldings does not maintain canonical equivalence. Furthermore, it is often undesirable to have differing behavior for caseless matching. Because language information is often not available when caseless matching is applied to strings, it also may not be clear which alternate to choose.

The Unicode case folding algorithm is defined to be simpler and more efficient than case mappings. It is context-insensitive and language-independent (except for the optional, alternate Turkic case foldings). As a result, there are a few rare cases where a caseless match does not match pairs of strings as expected; the most notable instance of this is for Lithuanian. In Lithuanian typography for dictionary use, an "i" retains its dot when a grave, acute, or tilde accent is placed above it. This convention is represented in Unicode by using an explicit combining dot above, occurring in sequence between the "i" and the respective accent. (See *Figure 7-2.*) When case folded using the default case folding algorithm, strings containing these sequences will still contain the combining dot above. In the unusual situation where case folding needs to be tailored to provide for these special Lithuanian dictionary requirements, strings can be preprocessed to remove any combining dot above characters occurring between an "i" and a subsequent accent, so that the folded strings will match correctly.

Where case distinctions are not important, other distinctions between Unicode characters (in particular, compatibility distinctions) are generally ignored as well. In such circumstances, text can be normalized to Normalization Form NFKC or NFKD after case folding, thereby producing a normalized form that erases both compatibility distinctions and case distinctions. However, such normalization should generally be done only on a restricted repertoire, such as identifiers (alphanumerics). See Unicode Standard Annex #15, "Unicode Normalization Forms," and Unicode Standard Annex #31, "Unicode Identifier and Pattern Syntax," for more information. For a summary, see "Equivalent Sequences" in *Section 2.2, Unicode Design Principles.*

Caseless matching is only an approximation of the language-specific rules governing the strength of comparisons. Language-specific case matching can be derived from the collation data for the language, where only the first- and second-level differences are used. For more information, see Unicode Technical Standard #10, "Unicode Collation Algorithm."

In most environments, such as in file systems, text is not and cannot be tagged with language information. In such cases, the language-specific mappings *must not* be used. Otherwise, data structures such as B-trees might be built based on one set of case foldings and used based on a different set of case foldings. This discrepancy would cause those data structures to become corrupt. For such environments, a constant, language-independent, default case folding is required.

Stability. The definition of case folding is guaranteed to be stable, in that any string of characters case folded according to these rules will remain case folded in Version 5.0 or later of the Unicode Standard. To achieve this stability, there are constraints on additions of case pairs for existing encoded characters. Typically, no new lowercase character will be added to the Unicode Standard as a casing pair of an existing upper- or titlecase character that

does not already have a lowercase pair. In exceptional circumstances, where lowercase characters must be added to the standard in a later version than the version in which the corresponding uppercase characters were encoded, such lowercase characters can only be defined as new case pairs with a corresponding change to case folding to ensure that they *case fold* to the old uppercase letters. See the subsection "Policies" in *Section B.6, Other Unicode Online Resources*.

Normalization and Casing

Casing operations as defined in *Section 3.13, Default Case Algorithms* are not guaranteed to preserve Normalization Forms. That is, some strings in a particular Normalization Form (for example, NFC) will no longer be in that form after the casing operation is performed. Consider the strings shown in the example in *Table 5-5*.

Table 5-5. Casing and Normalization in Strings

Original (NFC)	ǰ̣	<U+01F0 LATIN SMALL LETTER J WITH CARON, U+0323 COMBINING DOT BELOW>
Uppercased	Jˇ̣	<U+004A LATIN CAPITAL LETTER J, U+030C COMBINING CARON, U+0323 COMBINING DOT BELOW>
Uppercased NFC	J̣ˇ	<U+004A LATIN CAPITAL LETTER J, U+0323 COMBINING DOT BELOW, U+030C COMBINING CARON>

The original string is in Normalization Form NFC format. When uppercased, the *small j with caron* turns into an *uppercase J* with a separate *caron*. If followed by a combining mark below, that sequence is not in a normalized form. The combining marks have to be put in canonical order for the sequence to be normalized.

If text in a particular system is to be consistently normalized to a particular form such as NFC, then the casing operators should be modified to normalize after performing their core function. The actual process can be optimized; there are only a few instances where a casing operation causes a string to become denormalized. If a system specifically checks for those instances, then normalization can be avoided where not needed.

Normalization also interacts with case folding. For any string X, let `Q(X) = NFC(toCasefold(NFD(X)))`. In other words, `Q(X)` is the result of normalizing X, then case folding the result, then putting the result into Normalization Form NFC format. Because of the way normalization and case folding are defined, `Q(Q(X)) = Q(X)`. Repeatedly applying `Q` does not change the result; case folding is *closed* under canonical normalization for either Normalization Form NFC or NFD.

Case folding is not, however, closed under compatibility normalization for either Normalization Form NFKD or NFKC. That is, given `R(X) = NFKC(toCasefold(NFD(X)))`, there are some strings such that `R(R(X)) ≠ R(X)`. NFKC_Casefold, a derived property, is closed under both case folding and NFKC normalization. The property values for

NFKC_Casefold are found in DerivedNormalizationProps.txt in the Unicode Character Database.

5.19 Mapping Compatibility Variants

Identifying one character as a compatibility variant of another character (or sequence of characters) suggests that in many circumstances the first can be remapped to the second without the loss of any textual information other than formatting and layout. (See *Section 2.3, Compatibility Characters*.)

Such remappings or foldings can be done in different ways. In the case of compatibility decomposable characters, remapping occurs as a result of normalizing to the NFKD or NFKC forms defined by Unicode Normalization. Other compatibility characters which are not compatibility decomposable characters may be remapped by various kinds of folding; for example, KangXi radical symbols in the range U+2F00..U+2FDF might be substituted by the corresponding CJK unified ideographs of the same appearance.

However, such remapping should not be performed indiscriminately, because many of the compatibility characters are included in the standard precisely to allow systems to maintain one-to-one mappings to other existing character encoding standards. In such cases, a remapping would lose information that is important to maintaining some distinction in the original encoding.

Thus an implementation must proceed with due caution—replacing a character with its compatibility decomposition or otherwise folding compatibility characters together with ordinary Unicode characters may change not only formatting information, but also other textual distinctions on which some other process may depend.

In many cases there exists a visual relationship between a compatibility character and an ordinary character that is akin to a font style or directionality difference. Replacing such characters with unstyled characters could affect the meaning of the text. Replacing them with rich text would preserve the meaning for a human reader, but could cause some programs that depend on the distinction to behave unpredictably. This issue particularly affects compatibility characters used in mathematical notation. For more discussion of these issues, see the W3C specification, "Unicode in XML and other Markup Languages," and Unicode Technical Report #25, "Unicode Support for Mathematics."

In other circumstances, remapping compatibility characters can be very useful. For example, transient remapping of compatibility decomposable characters using NFKC or NFKD normalization forms is very useful for performing "loose matches" on character strings. See also Unicode Technical Standard #10, "Unicode Collation Algorithm," for the role of compatibility character remapping when establishing collation weights for Unicode strings.

Confusables. The visual similarities between compatibility variants and ordinary characters can make them confusable with other characters, something that can be exploited in possible security attacks. Compatibility variants should thus be avoided in certain usage domains, such as personal or network identifiers. The usual practice for avoiding compatibility variants is to restrict such strings to those already in Normalization Form NFKC; this practice eliminates any compatibility decomposable characters. Compatibility decompos-

able characters can also be remapped on input by processes handling personal or network identifiers, using Normalization Form NFKC.

This general implementation approach to the problems associated with visual similarities among compatibility variants, by focusing first on the remapping of compatibility decomposable characters, is useful for two reasons. First, the large majority of compatibility variants are in fact also compatibility decomposable characters, so this approach deals with the biggest portion of the problem. Second, it is simply and reproducibly implementable in terms of a well-defined Unicode Normalization Form.

Extending restrictions on usage to other compatibility variants is more problematical, because there is no exact specification of which characters are compatibility variants. Furthermore, there may be valid reasons to restrict usage of certain characters which may be visually confusable or otherwise problematical for some process, even though they are not generally considered to be compatibility variants. Best practice in such cases is to depend on carefully constructed and justified lists of confusable characters.

For more information on security implications and a discussion of confusables, see Unicode Technical Report #36, "Unicode Security Considerations" and Unicode Technical Standard #39, "Unicode Security Mechanisms."

5.20 Unicode Security

It is sometimes claimed that the Unicode Standard poses new security issues. Some of these claims revolve around unique features of the Unicode Standard, such as its encoding forms. Others have to do with generic issues, such as character spoofing, which also apply to any other character encoding, but which are seen as more severe threats when considered from the point of view of the Unicode Standard.

This section examines some of these issues and makes some implementation recommendations that should help in designing secure applications using the Unicode Standard.

Alternate Encodings. A basic security issue arises whenever there are alternate encodings for the "same" character. In such circumstances, it is always possible for security-conscious modules to make different assumptions about the representation of text. This conceivably can result in situations where a security watchdog module of some sort is screening for prohibited text or characters, but misses the same characters represented in an alternative form. If a subsequent processing module then treats the alternative form as if it were what the security watchdog was attempting to prohibit, one potentially has a situation where a hostile outside process can circumvent the security software. Whether such circumvention can be exploited in any way depends entirely on the system in question.

Some earlier versions of the Unicode Standard included enough leniency in the definition of the UTF-8 encoding form, particularly regarding the so-called *non-shortest form*, to raise questions about the security of applications using UTF-8 strings. However, the conformance requirements on UTF-8 and other encoding forms in the Unicode Standard have been tightened so that no encoding form now allows any sort of alternate representation, including non-shortest form UTF-8. Each Unicode code point has a single, unique encoding in any particular Unicode encoding form. Properly coded applications should not be subject to attacks on the basis of code points having multiple encodings in UTF-8 (or UTF-16).

However, another level of alternate representation has raised other security questions: the canonical equivalences between precomposed characters and combining character sequences that represent the same abstract characters. This is a different kind of alternate representation problem—not one of the encoding forms per se, but one of visually identical characters having two distinct representations (one as a single encoded character and one as a sequence of base form plus combining mark, for example). The issue here is different from that for alternate encodings in UTF-8. Canonically equivalent representations for the "same" string are perfectly valid and expected in Unicode. The conformance requirement, however, is that conforming implementations cannot be *required* to make an interpretation distinction between canonically equivalent representations. The way for a security-conscious application to guarantee this is to carefully observe the normalization specifications (see Unicode Standard Annex #15, "Unicode Normalization Forms") so that data is handled consistently in a normalized form.

Spoofing. Another security issue is *spoofing*, meaning the deliberate misspelling of a domain name, or user name, or other string in a form designed to trick unwary users into interacting with a hostile website as if it was a trusted site (or user). In this case, the confu-

sion is not at the level of the software process handling the code points, but rather in the human end users, who see one character but mistake it for another, and who then can be fooled into doing something that will breach security or otherwise result in unintended results.

To be effective, spoofing does not require an exact visual match—for example, using the digit "1" instead of the letter "l". The Unicode Standard contains many *confusables*—that is, characters whose glyphs, due to historical derivation or sheer coincidence, resemble each other more or less closely. Certain security-sensitive applications or systems may be vulnerable due to possible misinterpretation of these confusables by their users.

Many legacy character sets, including ISO/IEC 8859-1 or even ASCII, also contain confusables, albeit usually far fewer of them than in the Unicode Standard simply because of the sheer scale of Unicode. The legacy character sets all carry the same type of risks when it comes to spoofing, so there is nothing unique or inadequate about Unicode in this regard. Similar steps will be needed in system design to assure integrity and to lessen the potential for security risks, no matter which character encoding is used.

The Unicode Standard encodes characters, not glyphs, and it is impractical for many reasons to try to avoid spoofing by simply assigning a single character code for every possible confusable glyph among all the world's writing systems. By unifying an encoding based strictly on appearance, many common text-processing tasks would become convoluted or impossible. For example, Latin B and Greek Beta B look the same in most fonts, but lower-case to two different letters, Latin b and Greek beta β, which have very distinct appearances. A simplistic fix to the confusability of Latin B and Greek Beta would result in great difficulties in processing Latin and Greek data, and in many cases in data corruptions as well.

Because all character encodings inherently have instances of characters that might be confused with one another under some conditions, and because the use of different fonts to display characters might even introduce confusions between characters that the designers of character encodings could not prevent, character spoofing must be addressed by other means. Systems or applications that are security-conscious can test explicitly for known spoofings, such as "MICROS0FT," "A0L," or the like (substituting the digit "0" for the letter "O"). Unicode-based systems can provide visual clues so that users can ensure that labels, such as domain names, are within a single script to prevent cross-script spoofing. However, provision of such clues is clearly the responsibility of the system or application, rather than being a security condition that could be met by somehow choosing a "secure" character encoding that was not subject to spoofing. No such character encoding exists.

Unicode Standard Annex #24, "Unicode Script Property," presents a classification of Unicode characters by script. By using such a classification, a program can check that labels consist only of characters from a given script or characters that are expected to be used with more than one script (such as the "Common" or "Inherited" script names defined in Unicode Standard Annex #24, "Unicode Script Property"). Because cross-script names may be legitimate, the best method of alerting a user might be to highlight any unexpected boundaries between scripts and let the user determine the legitimacy of such a string explicitly.

For further discussion of security issues, see Unicode Technical Report #36, "Unicode Security Considerations," and Unicode Technical Standard #39, "Unicode Security Mechanisms."

5.21 Ignoring Characters in Processing

The majority of encoded characters in the Unicode Standard are ordinary graphic characters. However, the standard also includes a significant number of special-use characters. For example, format characters (General_Category=Cf) are often defined to have very particular effects in text processing. These effects may impact one kind of text process, but be completely irrelevant for other text processes. Format characters also typically have no visible display of their own, but may impact the display of neighboring graphic characters. Technically, variation selectors are not format characters, but combining marks. However, variation selectors and other "invisible" combining marks also have special behavior in text processing.

Other sections of the Unicode Standard specify the intended effects of such characters in detail. See, for example, *Section 23.2, Layout Controls* and *Section 23.4, Variation Selectors.* This section, on the other hand, approaches the issue by discussing which kinds of format characters (and other characters) are *ignored* for different kinds of text processes, and providing pointers to related implementation guidelines.

How these kinds of special-use characters are displayed or *not* displayed in various contexts is of particular importance. Many have no inherent display of their own, so pose questions both for normal rendering for display and for fallback rendering. Because of this, a particularly detailed discussion of ignoring characters for display can be found toward the end of this section.

Characters Ignored in Text Segmentation

Processing for text segmentation boundaries generally ignores certain characters which are irrelevant to the determination of those boundaries. The exact classes of characters depend on which type of text segmentation is involved.

When parsing grapheme cluster boundaries, characters used to extend grapheme clusters are ignored for boundary determination. These include nonspacing combining marks and enclosing marks, as well as U+200C ZERO WIDTH NON-JOINER. The exact list of characters involved is specified by the property value: Grapheme_Cluster_Break=Extend. U+200D ZERO WIDTH JOINER requires special handling, particularly for emoji sequences.

When parsing word or sentence boundaries, the set of characters which are ignored for boundary determination is enlarged somewhat, to include spacing combining marks and most format characters. For word breaking, the exact list of characters is specified by two property values: Word_Break=Extend or Word_Break=Format. For sentence breaking, the corresponding property values are: Sentence_Break=Extend or Sentence_Break=Format.

For a detailed discussion of text segmentation, see Unicode Standard Annex #29, "Unicode Text Segmentation." In particular, see Section 6.2, *Replacing Ignore Rules,* in that annex, for implementation notes about the rules which ignore classes of characters for segmentation.

Characters Ignored in Line Breaking

Most control characters and format characters are ignored for line break determination, and do not contribute to line width. The Unicode Line Breaking Algorithm handles this class of characters by giving them the same Line_Break property value as combining marks: Line_Break=CM. For a detailed discussion, see Unicode Standard Annex #14, "Unicode Line Breaking Algorithm."

When expanding or compressing intercharacter space, as part of text justification and determination of line breaks, the presence of U+200B ZERO WIDTH SPACE or U+2060 WORD JOINER is generally ignored. There are, however, occasional exceptions. See, for example, the discussion of "Thai-style" letter spacing in *Section 23.2, Layout Controls*.

Characters Ignored in Cursive Joining

U+200C ZERO WIDTH NON-JOINER and U+200D ZERO WIDTH JOINER are format controls specifically intended to influence cursive joining. However, there are other format controls which are explicitly ignored when processing text for cursive joining. In particular, U+2060 WORD JOINER, U+FEFF ZERO WIDTH NO-BREAK SPACE, and U+200B ZERO WIDTH SPACE influence text segmentation and line breaking, but should be ignored for cursive joining. U+034F COMBINING GRAPHEME JOINER is also ignored for cursive joining.

More generally, there is a broad class of characters whose occurrence in a string should be ignored when calculating cursive connections between adjacent letters subject to cursive joining. This class is defined by the property value, Joining_Type=Transparent, and includes all nonspacing marks and most format characters other than ZWNJ and ZWJ. See the detailed discussion of cursive joining in *Section 23.2, Layout Controls*.

Characters Ignored in Identifiers

Characters with the property Default_Ignorable_Code_Point (DI) are generally not recommended for inclusion in identifiers. Such characters include many (but not all) format characters, as well as variation selectors. Exceptions are the cursive joining format characters, U+200C ZERO WIDTH NON-JOINER and U+200D ZERO WIDTH JOINER, which in limited circumstances may be used to make visual distinctions deemed necessary for identifiers.

There are several possible approaches for ensuring that characters with DI=True are not significant for comparison of identifiers. A strict formal syntax definition may simply prohibit their inclusion in identifier strings altogether. However, comparison of identifiers often involves a folding operation, such as case folding. In applications which implement identifier folding based on the toNFKC_CaseFold transformation, DI=True characters are removed from a string by that transformation. With such an approach, DI=True characters can be said to be "ignored" in identifier comparison, and their presence or absence in a given identifier string is irrelevant to the comparison. See Unicode Standard Annex #31, "Unicode Identifier and Pattern Syntax," for a detailed discussion of normalization and case folding of identifiers and of the handling of format characters in identifiers.

Characters Ignored in Searching and Sorting

Searching and string matching is another context in which particular characters may be ignored. Typically, users expect that certain characters, such as punctuation, will be ignored when looking for string matches against a target string, or they expect that certain character distinctions, such as case differences, will be ignored. Exact binary string comparisons in such circumstances produce the wrong results.

At its core, sorting string data involves using a string matching algorithm to determine which strings count as equal. In any comparison of strings which do not count as equal, sorting additionally requires the ability to determine which string comes before and which after in the collation order. It is important to have a well-defined concept of which characters "do not make a difference," and are thus ignored for the results of the sorting.

Some Unicode characters almost never make a significant difference for searching, string matching, and sorting. For example, U+200C ZERO WIDTH NON-JOINER and U+200D ZERO WIDTH JOINER may impact cursive joining or ligature formation, but are not intended to represent semantic differences between strings. At a first level of approximation, most Unicode format controls should be ignored for searching and sorting. However, there is no unique way to use Unicode character properties to devise an exact list of which characters should always be ignored for searching and sorting, in part because the criteria for any particular search or sort can vary so widely.

The Unicode algorithm which addresses this issue generically is defined in Unicode Technical Standard #10, "Unicode Collation Algorithm." The Default Unicode Collation Element Table (DUCET), documented in that standard, provides collation weights for all Unicode characters; many of those weights are set up so that the characters will be ignored by default for sorting. A string matching algorithm can also be based on the weights in that table. Additionally, the UCA provides options for ignoring *distinctions* between related characters, such as uppercase versus lowercase letters, or letters with or without accents. The UCA provides a mechanism to tailor the DUCET. This mechanism not only enables the general algorithm to support different tailored tables which allow for language-specific orderings of characters, it also makes it possible to specify very precisely which characters should or should not be ignored for any particular search or sort.

Characters Ignored for Display

There are two distinct cases to consider when determining whether a particular character should be "ignored" for display. The first case involves normal rendering, when a process supports the character in question. The second case involves fallback rendering, when the character in question is outside the repertoire which can be supported for normal rendering, so that a fallback to exceptional rendering for unknown characters is required.

In this discussion, "display" is used as shorthand for the entire text rendering process, which typically involves a combination of rendering software and font definition. Having a display glyph for a character defined in a font is not sufficient to render it for screen display or for printing; rendering software is involved as well. On the other hand, fonts may con-

tain complex rendering logic which contributes to the text rendering process. This discussion is not meant to preclude any particular approach to the design of a full text rendering process. A phrase such as, "a font displays a glyph for the character," or "a font displays no glyph for the character," is simply a general way of describing the intended display outcome for rendering that character.

Normal Rendering. Many characters, including format characters and variation selectors, have no visible glyph or advance width directly associated with them. Such characters without glyphs are typically shown in the code charts with special display glyphs using a dotted box and a mnemonic label. (See *Section 24.1, Character Names List*, for code chart display conventions.) Outside of the particular context of code chart display, a font will typically display no glyph for such characters. However, it is not unusual for format characters and variation selectors to have a visible effect on other characters in their vicinity. For example, ZWJ and ZWNJ may affect cursive joining or the appearance of ligatures. A variation selector may change the choice of glyph for display of the base character it follows. In such cases, even though the format character or variation selector has no visible glyph of its own, it would be inappropriate to say that it is *ignored* for display, because the intent of its use is to change the display in some visible way. Additional cases where a format character has no glyph, but may otherwise affect display include:

- Bidirectional format characters do not affect the glyph forms of displayed characters, but may cause significant rearrangements of spans of text in a line.

- U+00AD `SHY` SOFT HYPHEN has a null default appearance in the middle of a line: the appearance of "ther`SHY`apist" is simply "therapist"—no visible glyph. In line break processing, it indicates a possible intraword break. At any intraword break that is used for a line break—whether resulting from this character or by some automatic process—a hyphen glyph (perhaps with spelling changes) or some other indication can be shown, depending on language and context.

In other contexts, a format character may have no visible effect on display at all. For example, a ZWJ might occur in text between two characters which are not subject to cursive joining and for which no ligature is available or appropriate: <x, ZWJ, x>. In such a case, the ZWJ simply has no visible effect, and one can meaningfully say that it is *ignored* for display. Another example is a variation selector following a base character for which no standardized or registered variation sequence exists. In that case, the variation selector has no effect on the display of the text.

Finally, there are some format characters whose function is not intended to affect display. U+200B ZERO WIDTH SPACE affects word segmentation, but has no visible display. U+034F COMBINING GRAPHEME JOINER is likewise always ignored for display. Additional examples include:

- U+2060 `WJ` WORD JOINER does not produce a visible change in the appearance of surrounding characters; instead, its only effect is to indicate that there should be no line break at that point.

- U+2061 $\overline{f()}$ FUNCTION APPLICATION has no effect on the text display and is used only in internal mathematical expression processing.

The fact that format characters and variation selectors have no visible glyphs does not mean that such characters must always be invisible. An implementation can, for example, show a visible glyph on request, such as in a "Show Hidden" mode. A particular use of a "Show Hidden" mode is to display a visible indication of misplaced or ineffectual format characters. For example, a sequence of two adjacent joiners, <..., ZWJ, ZWJ, ...>, is a case where the extra ZWJ should have no effect.

Format characters with no visible glyphs are different from space characters. Space characters, such as U+0020 SPACE, are classified as graphic characters. Although they do not have *visible* glyphs for display, they have advance widths. Technically, that counts as a "glyph" in a font—it is simply a blank glyph "with no pixels turned on." Like other graphic characters, a space character can be visibly selected in text. Line separation characters, such as the *carriage return*, do not clearly exhibit their advance width, because they always occur at the end of a line, but most implementations give them a visible advance width when they are selected. Hence, they are classed together with space characters; both are given the White_Space property. Whitespace characters are not considered to be ignored for display.

Fallback Rendering. Fallback rendering occurs when a text process needs to display a character or sequence of characters, but lacks the rendering resources to display that character correctly. The typical situation results from having text to display without an appropriate font covering the repertoire of characters used in that text. The recommended behavior for display in such cases is to fall back to some visible, but generic, glyph display for graphic characters, so that at least it is clear that there are characters present—and usually, how many are present. (See *Section 5.3, Unknown and Missing Characters.*) However, variation selectors and some format characters are special—it is not appropriate for fallback rendering to display them with visible glyphs. This is illustrated by the following examples.

First consider an ordinary graphic character. For example, if an implementation does not support U+0915 क DEVANAGARI LETTER KA, it should not ignore that character for display. Displaying *nothing* would give the user the impression that the character does not occur in the text at all. The recommendation in that case is to display a "last-resort" glyph or a visible "missing glyph" box, instead.

Contrast that with the typical situation for a format character, such as ZWJ. If an implementation does not support that character at all, the best practice is to ignore it completely for display, without showing a last-resort glyph or a visible box in its place. This is because even for normal rendering a ZWJ is invisible—its visible effects are on other characters. When an implementation does not support the behavior of a ZWJ, it has no way of showing the effects on neighboring characters.

Default Ignorable Code Point. The list of characters which should be ignored for display in fallback rendering is given by a character property: Default_Ignorable_Code_Point (DI). Those characters include almost all format characters, all variation selectors, and a few other exceptional characters, such as Hangul fillers. The exact list is defined in DerivedCoreProperties.txt in the Unicode Character Database.

The Default_Ignorable_Code_Point property is also given to certain ranges of code points: U+2060..U+206F, U+FFF0..U+FFF8, and U+E0000..U+E0FFF, including any *unassigned* code points in those ranges. These ranges are designed and reserved for future encoding of format characters and similar special-use characters, to allow a certain degree of forward compatibility. Implementations which encounter unassigned code points in these ranges should ignore them for display in fallback rendering.

Surrogate code points, private-use characters, and control characters are not given the Default_Ignorable_Code_Point property. To avoid security problems, such characters or code points, when not interpreted and not displayable by normal rendering, should be displayed in fallback rendering with a fallback glyph, so that there is a visible indication of their presence in the text. For more information, see Unicode Technical Report #36, "Unicode Security Considerations."

A small number of format characters (General_Category=Cf) are also not given the Default_Ignorable_Code_Point property. This may surprise implementers, who often assume that all format characters are generally ignored in fallback display. The exact list of these exceptional format characters can be found in the Unicode Character Database. There are, however, two important sets of such format characters to note. First, there are the visible format characters which span groups of numbers, particularly for the Arabic script—for example, U+0601 ARABIC SIGN SANAH, the Arabic year sign. These prepended concatenation marks always have a visible display. See "Signs Spanning Numbers" in *Section 9.2, Arabic* for more discussion of the use and display of these signs. The other notable set of exceptional format characters is the interlinear annotation characters: U+FFF9 INTERLINEAR ANNOTATION ANCHOR through U+FFFB INTERLINEAR ANNOTATION TERMINATOR. These annotation characters should have a visible glyph display for fallback rendering, because if they are simply not displayed, there is too much potential to misread the resulting displayed text. See "Annotation Characters" in *Section 23.8, Specials* for more discussion of the use and display of interlinear annotation characters.

5.22 Best Practice for U+FFFD Substitution

When converting text from one character encoding to another, a conversion algorithm may encounter unconvertible code units. This is most commonly caused by some sort of corruption of the source data, so that it does not correctly follow the specification for that character encoding. Examples include dropping a byte in a multibyte encoding such as Shift-JIS, improper concatenation of strings, a mismatch between an encoding declaration and actual encoding of text, use of non-shortest form for UTF-8, and so on.

When a conversion algorithm encounters such unconvertible data, the usual practice is either to throw an exception or to use a defined substitution character to represent the unconvertible data. In the case of conversion *to* one of the encoding forms of the Unicode Standard, the substitution character is defined as U+FFFD REPLACEMENT CHARACTER. However, there are different possible ways to use U+FFFD. This section describes the best practice.

For conversion *between* different encoding forms of the Unicode Standard, *Section 3.9, Unicode Encoding Forms* defines best practice for the use of U+FFFD. The basic formulation is as follows:

> **Whenever an unconvertible offset is reached during conversion of a code unit sequence:**
>
> **1. The maximal subpart at that offset should be replaced by a single U+FFFD.**
>
> **2. The conversion should proceed at the offset immediately after the maximal subpart.**

In that formulation, the term "maximal subpart" refers to a *maximal subpart of an ill-formed subsequence*, which is precisely defined in *Section 3.9, Unicode Encoding Forms* for Unicode encoding forms. Essentially, a conversion algorithm gathers up the longest sequence of code units that could be the start of a valid, convertible sequence, but which is not actually convertible. For example, consider the first three bytes of a four-byte UTF-8 sequence, followed by a byte which cannot be a valid continuation byte: <F4 80 80 41>. In that case <F4 80 80> would be the maximal subpart that would be replaced by a single U+FFFD. If there is not *any* start of a valid, convertible sequence in the unconvertible data at a particular offset, then the maximal subpart would consist of a single code unit.

This practice reflects the way conversion processes are typically constructed, particularly for UTF-8. An optimized conversion algorithm simply walks an offset down the source data string until it collects a sequence it can convert or until it reaches the first offset at which it knows it *cannot* convert that sequence. At that point it either throws an exception or it substitutes the unconvertible sequence it has collected with a single U+FFFD and then moves on to the next offset in the source.

Although the definition of best practice for U+FFFD substitution in *Section 3.9, Unicode Encoding Forms* technically applies only to conversion between Unicode encoding forms, that principle for dealing with substitution for unconvertible sequences can be extended

easily to cover the more general case of conversion of any external character encoding to Unicode. The more general statement is as follows:

> **Whenever an unconvertible offset is reached during conversion of a code unit sequence to Unicode:**
>
> **1. Find the longest code unit sequence that is the initial subsequence of some sequence that could be converted. If there is such a sequence, replace it with a single U+FFFD; otherwise replace a single code unit with a single U+FFFD.**
>
> **2. The conversion should proceed at the offset immediately after the subsequence which has been replaced.**

When dealing with conversion mappings from external character encodings to Unicode, one needs to take into account the fact that the mapping may be many-to-one. The conversion algorithm needs to find the *longest* sequence that is valid for conversion, so that it does not prematurely convert a code unit that could be part of a longer valid sequence. (This problem does not occur when converting between Unicode encoding forms, which are all constructed to be non-overlapping and one-to-one transforms.)

The requirement for finding the longest valid sequence for conversion is then generalized to the case of replacement of invalid sequences. The conversion should proceed as far as it can down the input string while the input could still be interpreted as starting *some* valid sequence. Then if the conversion fails, all of the code units that have been collected to that point are replaced with a single U+FFFD. If there is no valid code unit at all, a single code unit is replaced.

For legacy character encodings and other character encodings defined externally, the Unicode Standard cannot precisely specify what is well-formed or ill-formed. Therefore, best practice for U+FFFD substitution is defined in terms of what is convertible or unconvertible in particular cases. Ultimately, that depends on the content of character mapping tables and their accompanying conversion algorithms. To the extent that implementations share common character mapping tables, they can obtain interoperable conversion results, not only for the convertible data, but also for any data unconvertible by those tables. Unicode Technical Standard #22, "Character Mapping Markup Language," provides an XML format for precisely specifying character mapping tables, which can be used to help guarantee interoperable conversions.