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

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

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

The publisher offers discounts on this book when ordered in quantity for bulk purchases and special sales. For more information, customers in the U.S. please contact U.S. Corporate and Government Sales, (800) 382-3419, corpsales@pearsontechgroup.com. For sales outside of the U.S., please contact International Sales, +1 317 581 3793, international@pearsontechgroup.com

Visit Addison-Wesley on the Web: http://www.awprofessional.com

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

QA268.U545 2004
005.7'2—dc21

2003052158

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

Chapter 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 Transcoding to Other Standards

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.

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.

Issues

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 $O(n)$. Generally, tables—as opposed to algorithmic transformation—are required to map between the Unicode Standard and another standard. Table lookup often yields much better performance than even simple algorithmic conversions, such as can be implemented between JIS and Shift-JIS.

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. Such 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 frequency of usage among characters differs widely and 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 large sections of the large-sized reverse mapping tables (containing the default character, or unmappable character entry) unused—and therefore paged to disk.

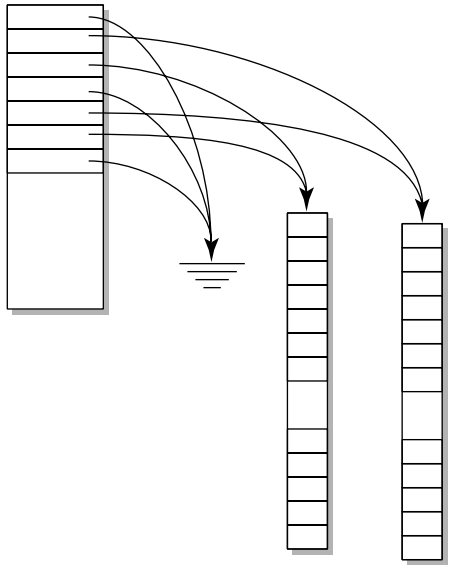
Ranges. It may be tempting to "optimize" these tables for space by providing elaborate provisions for nested ranges or similar devices. This practice leads to unnecessary performance costs on modern, highly pipelined processor architectures because of branch penalties. A faster solution is to use an *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 for a lookup 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, so that 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.

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 simply 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.

Figure 5-1. Two-Stage Tables



5.2 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 does not have a font for it or is otherwise incapable of rendering it appropriately.

In this case, an implementation might be able to provide further 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 , , ,  ,  ,  ,  ,  ,  , , and so on.

Default Property Values

To work properly in implementations, unassigned code points must be given default property values as if they were characters, because various algorithms require property values to be assigned to every code point to function at all. These default values are not uniform across all unassigned code points, because certain ranges of code points need different values to maximize compatibility with expected future assignments. For information on the default values for each property, see its description in the Unicode Character Database.

Except where indicated, the default values are not normative—conformant implementations can use other values. For example, instead of using the defined default values, an implementation might chose to interpolate the property values of assigned characters bordering a range of unassigned characters, using the following rules:

- Look at the nearest assigned characters in both directions. If they are in the same block and have the same property value, then use that value.

- From any block boundary, extending to the nearest assigned character inside the block, use the property value of that character.
- For all code points entirely in empty or unassigned blocks, use the default property value for that property.

There are two important benefits of using that approach in implementations. Property values become much more contiguous, allowing better compaction of property tables using structures such as a trie. (For more information on multistage tables, see *Section 5.1, Transcoding to Other Standards*.) Furthermore, because similar characters are often encoded in proximity, chances are good that the interpolated values will match the actual property values when characters are assigned to a given code point later.

Default Ignorable Code Points

Normally, code points outside the repertoire of supported characters would be displayed with a fallback glyph, such as a black box. However, format and control characters must not have visible glyphs (although they may have an effect on other characters in display). These characters are also ignored except with respect to specific, defined processes; for example, ZERO WIDTH NON-JOINER is ignored in collation. To allow a greater degree of compatibility across versions of the standard, the ranges U+2060..U+206F, U+FFF0..U+FFFB, and U+E0000..U+E0FFF are reserved for format and control characters (General Category = Cf). Unassigned code points in these ranges should be ignored in processing and display. For more information, see *Section 5.20, Default Ignorable Code Points*.

Interacting with Downlevel Systems

Versions of the Unicode Standard after Unicode 2.0 are strict supersets of earlier 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, a 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 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 is 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. However, the individual *components* of implementations may have different levels of support for surrogates, as long as those components are assembled and communicate correctly. The different levels of support are based on two primary issues:

- Does the implementation interpret supplementary characters?
- Does the implementation guarantee the integrity of a surrogate pair?

Various choices give rise to four possible levels of support for surrogate pairs in UTF-16, as shown in *Table 5-1*.

Table 5-1. Surrogate Support Levels

Support Level	Interpretation	Integrity of Pairs
None	No supplementary characters	Does not guarantee
Transparent	No supplementary characters	Guarantees
Weak	Some supplementary characters	Does not guarantee
Strong	Some supplementary characters	Guarantees

Without surrogate support, an implementation would not interpret any supplementary characters, and would not 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.

Transparent surrogate support applies to such components as encoding form conversions, which might fully guarantee the correct handling of surrogate pairs, but which in themselves do not interpret any supplementary characters. It also applies to components that handle low-level string processing, where a Unicode string is not interpreted but is handled simply as an array of code units irrespective of their status as surrogates. 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.

Weak surrogate support—that is, handling only those surrogate pairs correctly that correspond to interpreted characters—may be an appropriate design where the calling components are guaranteed not to pass uninterpreted characters. A rendering system, for example, might not be set up to deal with arbitrary surrogate pairs, but may still function correctly as long as its input is restricted to supported characters.

Components with mixed levels of surrogate support, if used correctly when integrated into larger systems, are consistent with an implementation as a whole having full surrogate support. It is important for each component of such a mixed system to have a robust implementation, so that the components providing full surrogate support are prepared to deal with the consequences of modules with no surrogate support occasionally "getting it wrong" and violating surrogate pair integrity. Robust UTF-16 implementations should not choke and die if they encounter isolated surrogate code units.

Example. The following sentence could be displayed in several different ways, depending on the level of surrogate support and the availability of fonts: "The Greek letter delta Δ is unrelated to the Ugaritic letter delta 𐎄." In UTF-16, the supplementary character for Ugaritic would, of course, be represented as a surrogate pair: <DC00 DB84>. The ■ in *Table 5-2* represents any visual representation of an unrenderable character by the implementation.

Table 5-2. Surrogate Level Examples

None	"The Greek letter delta Δ is unrelated the Ugaritic letter delta ■ ■."
Strong (glyph missing)	"The Greek letter delta Δ is unrelated to the Ugaritic letter delta ■."
Strong (glyph available)	"The Greek letter delta Δ is unrelated to the Ugaritic letter delta 𐎄."

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," much as "ij" in Dutch or "ll" in traditional Spanish.
- 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, "Text Boundaries."

Users are prevented from damaging the text if a text editor keeps *insertion points* (also known as *carets*) on character boundaries. As with text-element boundaries, the lowest-level string-handling routines (such as `wcschr`) do not necessarily need to be modified to prevent surrogates from being damaged. In practice, it is sufficient that only certain higher-level processes (such as those just noted) be aware of surrogate pairs; the lowest-level routines can continue to function on sequences of 16-bit code units (Unicode strings) without having to treat surrogates specially.

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 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. Both of them are appropriate only for positive integer writing.

It is also possible to write numbers in two ways with ideographic digits. For example, *Figure 5-2* shows how the number 1,234 can be written.

Figure 5-2. Ideographic Numbers

一千二百三十四

or

一二三四

Supporting these digits for numerical parsing means that implementations must be smart about distinguishing between these 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 *Section 5.15, Identifiers*).

It is only at a second level (for example, when implementing a full mathematical formula parser) that considerations such as superscripting become crucial for interpretation.

5.6 Normalization

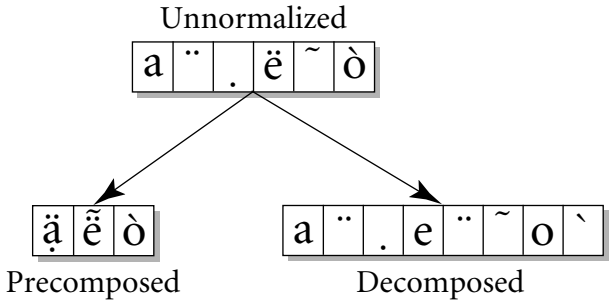
Alternative Spellings. The Unicode Standard contains explicit codes for the most frequently used accented characters. These characters can also be composed; in the case of accented letters, characters can be composed from a base character and nonspacing mark(s).

The Unicode Standard provides decompositions for characters that can be composed using a base character plus one or more nonspacing marks. Implementations that are “liberal” in what they accept, but “conservative” in what they issue, will have the fewest compatibility problems.

The decomposition mappings are specific to a particular version of the Unicode Standard. Additional mappings may result from character additions in future versions. See "Policies" in *Section B.4, Other Unicode References*, for more information.

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 5-3*).

Figure 5-3. 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; most existed in dominant standards.

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

The standard forms for normalization are defined in Unicode Standard Annex #15, "Unicode Normalization Forms." For further information see *Chapter 3, Conformance*; "Equivalent Sequences" in *Section 2.2, Unicode Design Principles*; and *Section 2.10, 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. Compression always constitutes a higher-level protocol and makes interchange dependent on knowledge of the compression method employed. For a detailed discussion on 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 transformation format or with ASCII codes. On the other hand, text consisting of ideographs encoded with UTF-8 will require more space than equivalent text encoded with UTF-16.

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, "Line Breaking Properties." Newlines are also handled specially in the context of regular expressions. For information, see Unicode Technical Report #18, "Unicode Regular Expression Guidelines." For the use of these characters in markup languages, see Unicode Technical Report #20, "Unicode in XML and Other Markup Languages."

Definitions

Table 5-3 provides hexadecimal values for the acronyms used in these guidelines.

Table 5-3. Hex Values for Acronyms

Acronym	Name	Unicode	ASCII	EBCDIC	
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

The acronyms shown in *Table 5-3* 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.

The Unicode Standard does not formally assign control characters; instead, it provides 65 corresponding code points for use as in various 7- and 8-bit character encoding standards. This enables correlations and cross-mappings between the Unicode Standard and other encodings as shown in the table. For more information, see *Section 15.1, Control Codes.*

For clarity, 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.

Table 5-3 shows that there are two mappings of LF and NEL used by EBCDIC systems. The first EBCDIC column shows the MVS Open Edition (including Code Page 1047) mapping of these characters. That mapping arises from the use of the LF character as "New Line" in ASCII-based Unix environments and in some data transfer protocols that use the Unix assumptions in an EBCDIC environment. The second column shows the Character Data Representation Architecture (CDRA) mapping, which is based on the standard definitions—both in ASCII and in EBCDIC—of LF.

NEL (next line) is not actually defined in ASCII. It is defined in the ISO control function standard, ISO 6429, as a C1 control function. However, the 0x85 mapping shown in *Table 5-3* is the usual way that this C1 control function is mapped in ASCII-based character encodings.

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 5-4*.

Table 5-4. NLF Platform Correlations

MacOS	CR
Unix	LF
Windows	CRLF
EBCDIC-based OS	NEL

Background

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
, 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 to 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 or 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 rules 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.

Rule 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.

R2c In parsing, choose the safest interpretation.

For example, in rule R2c an implementer dealing with sentence break heuristics would reason in the following way that it is safer to interpret any *NLF* as a 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 appropriately, 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 conventions just discussed in "Converting to Other Character Code Sets."***

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 should be independent of paragraph separation: A paragraph can start on one page and continue on the next page. Except when displaying on pages, it is interpreted in the same way as the LS in most parsing and in `readline`.

5.9 Regular Expressions

Byte-oriented regular expression engines require extensions to handle Unicode successfully. 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 Report #18, "Unicode Regular Expression Guidelines."

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 Spanish word "churo" after "czar," where it would be collated in traditional Spanish, nor would an English atlas put the Swedish city of Örebro after Zanzibar, where it would appear in Swedish.

However, language information is very 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. Although language information is 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.

Language information can be presented as out-of-band information or inline tags. In internal implementations, it is quite common to use out-of-band information, which is stored in data structures that are parallel to the text, rather than embedded in it. Out-of-band information does not interfere with the normal processing of the text (comparison, searching, and so on) and more easily supports manipulation of the text.

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

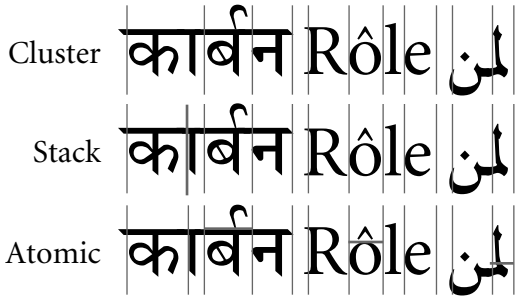
Consistent Text Elements

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 2.10, 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 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.

Figure 5-4. Consistent 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 5-4.*)

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 5-4*) can be selected only as a single unit. Stacked boundaries treat default grapheme clusters as single entities, much like composite characters. (See Unicode Standard Annex #29, "Text Boundaries," 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 linear 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.

Notice that 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 systems, the character is the smallest addressable item in text, so the selection and assignment of properties (such as font, color, letterspacing, and so on) are done on a per-character basis. There is no good way to simulate this addressability with precomposed characters. Systematically modifying all text editing to address parts of characters would be quite inefficient.

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.)

A relatively small number of implementation features are needed to support nonspacing marks. Different possible 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 languages, such as Arabic, Hebrew, and the languages of the Indian subcontinent, 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. A fixed set of composite character sequences can be rendered effectively by means of fairly simple substitution. Wherever a sequence of base character plus one or more nonspacing combining marks occurs, a glyph representing the combined form can be substituted. 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. When truncating strings, it is always easiest to truncate starting from the end and working backward. A trailing nonspacing mark will then not be separated from the preceding base character.

A more sophisticated rendering system can take into account more subtle variations in widths and kerning with nonspacing marks or account for those cases where the composite character sequence has a different advance width than the base character. Such rendering systems are not necessary for most applications. They can, however, also supply more sophisticated truncation routines. (See also *Section 5.13, Rendering Nonspacing Marks.*)

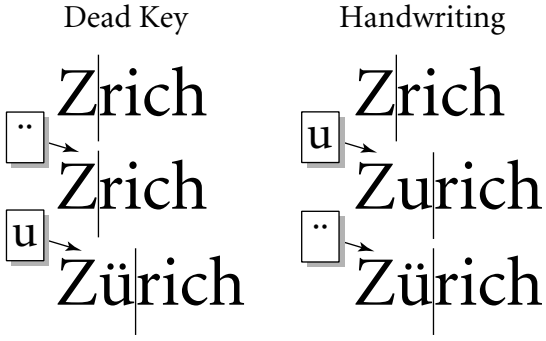
Other Processes. Correct multilingual comparison routines must already be able to compare a sequence of characters as one character, or one character as if it were a sequence. Such routines can also handle composite character sequences when supplied with the appropriate data. When searching strings, 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 5.16, Sorting and Searching; Section 5.6, Normalization;* and *Section 5.14, Locating Text Element Boundaries.*)

Keyboard Input

A common implementation for the input of composed character sequences is the use of so-called *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 composed character sequences or precomposed characters as needed. Although typists, especially in the Latin script, are trained on systems that work in this way, 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 5-5*).

Figure 5-5. 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.

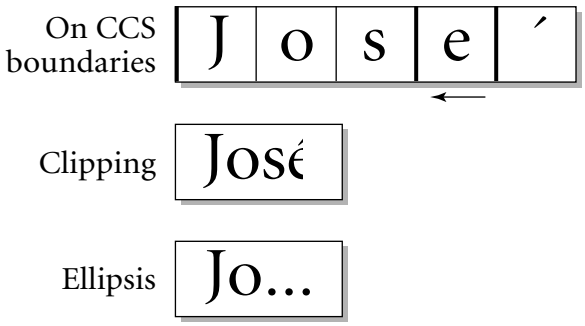
Truncation

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 length, some information will be lost. Truncating at a text element boundary (for example, on the last composite character sequence boundary or even last word boundary) is often preferable to truncating after the last code point, as shown in *Figure 5-6*. (See Unicode Standard Annex #29, "Text Boundaries.")

Figure 5-6. Truncating Composed Character Sequences



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 composed 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.

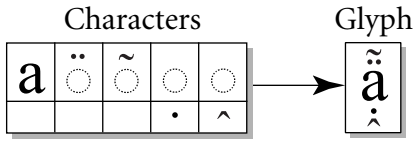
A different option is simply to clip the characters graphically. Unfortunately, the result may look ugly. Also, if the clipping occurs between characters, it may not give any visual feedback that characters are being omitted. A graphic or ellipsis can be used to give this visual feedback.

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, but in general, 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 2.10, Combining Characters*). This default treatment of multiple, potentially interacting nonspacing marks is known as the inside-out rule (see *Figure 5-7*).

Figure 5-7. 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 5-8*). 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 *Chapter 16, Code Charts*.)

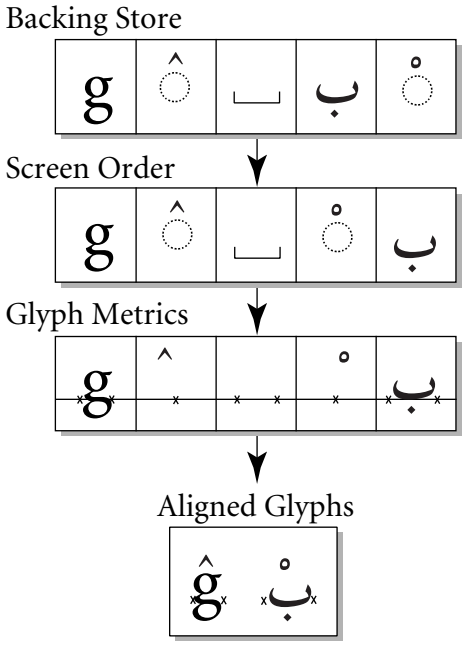
Figure 5-8. Fallback Rendering

Another method (*Simple Overlap*) uses default fixed positioning for an overlapping zero-width nonspacing mark, generally placed far away from possible base characters. 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 formatting 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 space as a base character. (See *Section 7.7, 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 5-9*). There are a few ways to accomplish this positioning.

Figure 5-9. Bidirectional Placement



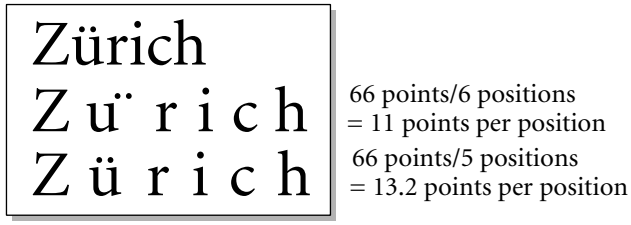
The simplest method is similar to the *Simple Overlap* fallback method. In the bidirectional algorithm, combining marks take the level of their base character. In that case, Arabic and Hebrew nonspacing marks would come to the left of their base characters. The font is designed so that instead of overlapping to the left, the Arabic and Hebrew nonspacing marks overlap to the right. In *Figure 5-9*, 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.

With some rendering software, it may be necessary to keep 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-left (odd) levels. (See Unicode Standard Annex #9, "The Bidirectional Algorithm.")

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 5-10*). This process needs to be modified if

zero-width nonspacing marks are present in the text. Otherwise, the nonspacing marks will be separated from their base characters.

Figure 5-10. Justification



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 the example in *Table 5-5*.

Table 5-5. 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 above. 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 5-6*.

Table 5-6. 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 *only* be permuted, not be combined or split. This can be restated as follows:

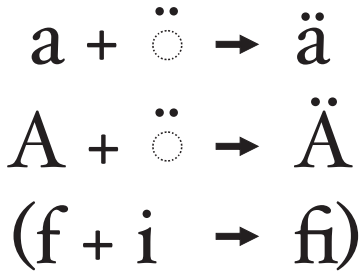
- Two characters that have the same combining class values cannot be given distinct internal weights.
- Two characters that have distinct combining class values cannot be given the same internal weight.
- Characters with a combining class of zero must be given an internal weight of zero.

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 composed character sequences can be rendered effectively by means of fairly simple substitution (see *Figure 5-11*). 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 further measures to account for those cases where the composed character sequence kerns differently or has a slightly different advance width than the base character.

Figure 5-11. Positioning with Ligatures



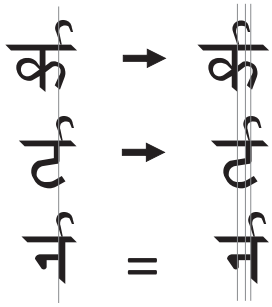
Positioning with ligatures is perhaps the simplest method of supporting nonspacing marks. Whenever there is a small, fixed set, such as those corresponding to the precomposed characters of ISO/IEC 8859-1 (Latin-1), this method is straightforward to apply. Because the composed character sequence almost always has the same width as the base character, ren-

dering, measurement, and editing of these characters are much easier than for the general case of ligatures.

If a composed character sequence does not form a ligature, then one of the two following methods 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 5-12*). In this case, several different glyphs correspond to different positions of the accents. Base glyphs generally fall into a fairly small number of classes, based on their general shape and width. According to the class of the base glyph, a particular glyph is chosen for a nonspacing mark.

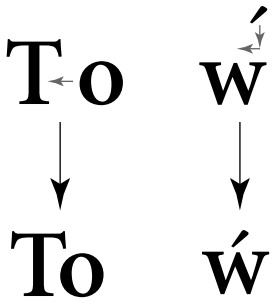
Figure 5-12. Positioning with Contextual Forms



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 5-13*). 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 5-13. Positioning with Enhanced Kerning



For effective use in the general case, the kerning process must also 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 programmatically. 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. A precise definition of the default Unicode mechanisms for determining such text element boundaries is found in Unicode Standard Annex #14, "Line Breaking Properties," and in Unicode Standard Annex #29, "Text Boundaries."

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 here as a recommended default for the definition of identifier syntax. These guidelines are no more complex than current rules in the common programming languages, except that they include more characters of different types.

Property-Based Identifier Syntax

The formal syntax provided here is intended to capture the general intent that an identifier consists of a string of characters that begins with a letter or an ideograph, and then includes any number of letters, ideographs, digits, or underscores. Each programming language standard has its own identifier syntax; different programming languages have different conventions for the use of certain characters from the ASCII range ($, @, #, _) in identifiers. To extend such a syntax to cover the full behavior of a Unicode implementation, implementers need only combine these specific rules with the sample syntax provided here.

The innovations in the sample identifier syntax to cover the Unicode Standard correctly include the following:

- Incorporation of proper handling of combining marks
- Allowance for layout and format control characters, which should be ignored when parsing identifiers

Combining Marks. Combining marks must be accounted for in identifier syntax. A composed character sequence consisting of a base character followed by any number of combining marks must be valid for an identifier. This requirement results from the conformance rules in *Chapter 3, Conformance*, regarding interpretation of canonical-equivalent character sequences.

Enclosing combining marks (for example, U+20DD..U+20E0) are excluded from the syntactic definition of <ident_extend>, because the composite characters that result from

their composition with letters (for example, U+24B6 CIRCLED LATIN CAPITAL LETTER A) are themselves not valid constituents of these identifiers.

Layout and Format Control Characters. The Unicode characters that are used to control joining behavior, bidirectional ordering control, and alternative formats for display are explicitly defined as not affecting breaking behavior. Unlike space characters or other delimiters, they do not serve to indicate word, line, or other unit boundaries. Accordingly, they are explicitly included for the purposes of identifier definition. Some implementations may choose to filter out these ignorable characters; this approach offers the advantage that two identifiers that appear to be identical are more likely to *be* identical.

Specific Character Adjustments. Specific identifier syntaxes can be treated as slight modifications of the generic syntax based on character properties. For example, SQL identifiers allow an underscore as an identifier part (but not as an identifier start); C identifiers allow an underscore as either an identifier part or an identifier start.

A useful set of characters to consider for exclusion from identifiers consists of all characters whose compatibility mappings have a `<font>` tag.

For the notation used in this section, see *Section 0.3, Notational Conventions*.

Syntactic Rule

<identifier> := <identifier_start> (<identifier_start> | <identifier_extend>)*

Identifiers are defined by a set of character categories from the Unicode Character Database. See *Table 5-7*.

Table 5-7. Syntactic Classes for Identifiers

Syntactic Class	Properties	Coverage
<identifier_start>	General Category = L or Nl, or Other_ID_Start = true	Uppercase letter, lowercase letter, titlecase letter, modifier letter, other letter, letter number, stability extensions
<identifier_extend>	General Category = Mn, Mc, Nd, Pc, or Cf	Nonspacing mark, spacing combining mark, decimal number, connector punctuation, formatting code

Backward Compatibility. Unicode General Category values are kept as stable as possible, but they can change across versions of the Unicode Standard. The Other_ID_Start property contains a small list of characters that qualified as identifier_start characters in some previous version of Unicode solely on the basis of their General Category properties, but that no longer qualify in the current version. In Unicode 4.0, this list consists of four characters:

U+2118 SCRIPT CAPITAL P

U+212E ESTIMATED SYMBOL

U+309B KATAKANA-HIRAGANA VOICED SOUND MARK

U+309C KATAKANA-HIRAGANA SEMI-VOICED SOUND MARK

The Other_ID_Start property is thus designed to ensure that the Unicode identifier specification is backward compatible: Any sequence of characters that qualified as an identifier in some version of Unicode will continue to qualify as an identifier in future versions.

Normalization. For programming language identifiers, normalization has a number of important implications. For a discussion of these issues, see "Annex 7: Programming Language Identifiers" in UAX #15, "Unicode Normalization Forms."

Alternative Recommendation

The down side of working with the syntactic classes defined in *Table 5-7* is the storage space needed for the detailed definitions, plus the fact that with each new version of the Unicode Standard new characters are added, which an existing parser would not be able to recognize. In other words, the recommendations based on that table are not upwardly compatible.

One method to address this problem is to turn the question around. Instead of defining the set of code points that are *allowed*, define a small, fixed set of code points that are reserved for syntactic use and allow everything else (including unassigned code points) as part of an identifier. All parsers written to this specification would behave the same way for all versions of the Unicode Standard, because the classification of code points is fixed forever.

The drawback of this method is that it allows "nonsense" to be part of identifiers because the concerns of lexical classification and of human intelligibility are separated. Human intelligibility can, however, be addressed by other means, such as usage guidelines that encourage a restriction to meaningful terms for identifiers. For an example of such guidelines, see the XML 1.1 specification by the W3C.

By increasing the set of *disallowed* characters, a somewhat intuitive recommendation for identifiers can be achieved. This approach uses the full specification of identifier classes, as of a particular version of the Unicode Standard, and permanently *disallows* any characters not recommended in that version for inclusion in identifiers. All code points unassigned as of that version would be *allowed* in identifiers, so that any future additions to the standard would already be accounted for. This approach ensures both upwardly compatible identifier stability and a reasonable division of characters into those that do and do not make human sense as part of identifiers.

Some additional extensions to the list of disallowed code points can be made to further constrain "unnatural" identifiers. For example, one could include unassigned code points in blocks of characters set aside for future encoding as symbols, such as mathematical operators.

With or without such fine-tuning, such a compromise approach still incurs the expense of implementing large lists of code points. While they no longer change over time, it is a matter of choice whether the benefit of enforcing somewhat word-like identifiers justifies their cost.

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 -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 LATIN CAPITAL LETTER A WITH DIAERESIS 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 in an order so 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 specifiying 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, there are some subtle differences 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 result in 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 further 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 diacriticals, 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, "Text Boundaries," 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*.

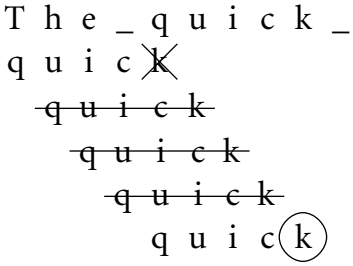
A related issue can arise because of inaccurate mappings from external character sets. To deal with this problem, characters that are easily confused by users can be kept in a weak equivalency class (đ *d-bar,* ð *eth,* Đ *capital d-bar,* Ð *capital eth*). This approach tends to do a better job of meeting 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 5-14*).

Figure 5-14. Sublinear Searching



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*, and in Unicode Technical Standard #10, "Unicode Collation Algorithm." However, there are many circumstances where 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 sort in code point order, while UTF-16 data (for code points higher than U+FFFF) does not. 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 only once per string is it necessary to do any extra work, not once per byte. That work can consistent 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, ...,                                          0x2F,
   0xD0, ...,                                          0xDF,
   0xE0, ...,                                0xED, 0xF0, 0xF1,
   0xF2, 0xF3, 0xF4, 0xEE, 0xEF, 0xF5, ...,            0xFF};
```

The rotate array is formed by taking an array of 256 bytes from 0x00 to 0xFF, and rotating 0xEE and 0xEF to a position after the bytes 0xF0..0xF4. 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 (ac + utf16Fixup[ac>>11]) -
      (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 the 32nd chunk correspond to the values 0xE000..0xF000. The addition of 0x2000 to the surrogate code units rotates them up to the range 0xF800..0xFFFF. Adding 0xF800 to the values 0xE000..0xF000 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 whenever it is 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 7.5, Georgian*, for more information.

Because of the inclusion of certain composite characters for compatibility, such as U+01F1 "DZ" LATIN CAPITAL LETTER DZ, there is a third case, called titlecase, which is used where the first character of a word is to be capitalized. An example of such a character is U+01F2 "Dz" LATIN CAPITAL LETTER D WITH SMALL LETTER Z.

Thus the three case forms are UPPERCASE, Titlecase, and lowercase.

The term "titlecase" can also be used to refer to words where the first letter is an uppercase or titlecase letter, and the rest of the letters are lowercase. However, not all words in the title of a document or first words in a sentence will be titlecase.

The choice of which words to titlecase is language-dependent. For example, "Taming of the Shrew" would be the appropriate capitalization in English, but not "Taming Of The Shrew." Moreover, the determination of what actually constitutes a word is language-dependent. For example, *l'arbre* might be considered two words in French, while *can't* is considered one word in English.

The case mappings in the Unicode Character Database (UCD) are informative, default mappings. Case itself, on the other hand, has normative status. For example, U+0041 "A" LATIN CAPITAL LETTER A is normatively uppercase, but its lowercase mapping to U+0061 "a" LATIN SMALL LETTER A is informative. This is because case can be considered an inherent property of a particular character, but case mappings between characters are occasionally influenced by local conventions.

Complications for Case Mapping

There are a number of complications to case mappings that occur once the repertoire of characters is expanded beyond ASCII.

In most cases, the titlecase is the same as the uppercase, but not always. For example, the titlecase of U+01F1 "DZ" *capital dz* is U+01F2 "Dz" *capital d with small z*.

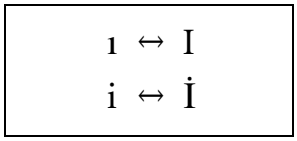
Case mappings may produce strings of different length than the original. For example, the German character U+00DF ß LATIN SMALL LETTER SHARP S expands when uppercased to the sequence of two characters "SS". This also occurs where there is no precomposed character corresponding to a case mapping, such as with U+0149 ŉ LATIN SMALL LETTER N PRECEDED BY APOSTROPHE.

There are some characters that require special handling, such as U+0345 *combining iota subscript*. As discussed in *Section 7.2, Greek*, the iota-subscript characters used to represent ancient text can be viewed as having special case mappings. Normally, the uppercase and lowercase forms of alpha-iota-subscript will map back and forth. In some instances, where 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).

Characters may also have different case mappings, depending on the context. 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.

Characters may have case mappings that depend on the locale. The principal example is Turkish, where U+0131 "ı" LATIN SMALL LETTER DOTLESS I maps to U+0049 "I" LATIN CAPITAL LETTER I and U+0069 "i" LATIN SMALL LETTER I maps to U+0130 "İ" LATIN CAPITAL LETTER I WITH DOT ABOVE, as shown in *Figure 5-15*.

Figure 5-15. Case Mapping for Turkish I



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.

Reversibility

It is important to note that 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*, 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. 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 strings to a canonical form where case differences 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 "κόσμος" and "ΚΟΣΜΟΣ" 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.***

Each equivalence class is completely disjoint from all the others, and every Unicode character is in one equivalence class. From each class, one representative element (a single lowercase letter where possible) is chosen to be the common form. CaseFolding.txt thus contains the mappings from other characters in the equivalence classes to their common forms.

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 KC or KD after case folding, to produce 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," 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. This discrepancy will cause those data structures to become corrupt. For such environments, a constant, language-independent, default case folding is required.

Normalization

Casing operations as defined below do not preserve normalization form. 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 5-8*.

Table 5-8. Casing and Normalization in Strings

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

The original string is in 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, it is denormalized. 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 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 either NFC or NFD.

Case folding is not, however, closed under compatibility normalization—either NFKD or NFKC. That is, given `R(X) = NFKC(toCasefold(NFD(X)))`, there are some strings such that `R(R(X)) ≠ R(X)`. A derived property, FC_NFKC_Closure, contains the additional mappings that can be used to produce a compatibility-closed case folding. This set of mappings is found in DerivedNormalizationProps.txt in the Unicode Character Database.

5.19 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 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 lead to questions regarding 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 or user name or other string in a form designed to trick unwary users into interacting with a hostile Web site as if it was a trusted site (or user). In this case, the confusion 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 lowercase 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, "Script Names," 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, "Script Names"). 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.

5.20 Default Ignorable Code Points

Default ignorable code points are those that should be ignored by default in rendering unless explicitly supported. They have no visible glyph or advance width in and of themselves, although they may affect the display, positioning, or adornment of adjacent or surrounding characters. Some default ignorable code points are assigned characters, while others are reserved for future assignment.

The default ignorable code points are listed in DerivedCoreProperties.txt in the Unicode Character Database with the property Default_Ignorable_Code_Points. Examples of such characters include U+2060 WORD JOINER, U+00AD SOFT HYPHEN, and U+200F RIGHT-TO-LEFT MARK.

An implementation should ignore default ignorable characters in rendering whenever it does *not* support the characters.

This can be contrasted with the situation for non-default ignorable characters. If an implementation does not support U+0915 क DEVANAGARI LETTER KA, for example, it should not ignore it in rendering. Displaying *nothing* would give the user the impression that it 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. See *Section 5.3, Unknown and Missing Characters*, for more information.

With default ignorable characters, such as U+200D ZWJ ZERO WIDTH JOINER, the situation is different. If the program does not support that character, the best practice is to ignore it completely without displaying a last-resort glyph or a visible box because the normal display of the character is invisible: Its effects are on other characters. Because the character is not supported, those effects cannot be shown.

Other characters will have other effects on adjacent characters. For example:

- 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 *f*() FUNCTION APPLICATION has no effect on the text display, and is used only in internal mathematical expression processing.
- 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.

This does *not* imply that default ignorable code points 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 show a visible indication of "misplaced" or "ineffectual" formatting codes. For example, this would include two adjacent U+200D ZWJ ZERO WIDTH JOINER characters, where the extra character has no effect.

The default ignorable *unassigned* code points lie in particular designated ranges. These ranges are designed and reserved for future default ignorable characters, to allow forward compatibility. All implementations should ignore all unassigned default ignorable code points in all rendering. Any new default ignorable characters should be assigned in those ranges, permitting existing programs to ignore them until they are supported in some future version of the program.

Some other characters have no visible glyphs—the whitespace characters. They typically have advance width, however. The line separation characters, such as the carriage return, do not clearly exhibit this advance width because they are always at the end of a line, but most implementations give them a visible advance width when they are selected.

Stateful Format Controls. There are a small number of *paired stateful controls*. These characters are used in pairs, with an initiating character (or sequence) and a terminating character. Even when these characters are ignored, complications can arise due to their paired nature. When text is deleted, these characters can become unpaired. To avoid this problem, any unpaired characters should be moved outside of the deletion so that the pairing is maintained. When text is copied or extracted, unpaired characters may also require the addition of the appropriate pairs to the copied text to maintain the pairing.

The paired stateful controls are listed in *Table 5-9*.

Table 5-9. Paired Stateful Controls

Characters	Documentation
Bidi Overrides and Embeddings	*Section 15.2, Layout Controls*; UAX #9
Deprecated Format Characters	*Section 15.4, Deprecated Format Characters*
Annotation Characters	*Section 15.9, Specials*
Tag Characters	*Section 15.10, Tag Characters*

The bidirectional overrides and embeddings and the annotation characters are more robust because their behavior terminates at paragraphs. The tag characters, on the other hand, are particularly fragile. See *Section 5.10, Language Information in Plain Text*, for more information.

Some other characters have a scope of influence over the behavior or rendering of neighboring characters. These include the *fraction slash* and the *arabic end of ayah*. However, because these characters are not paired, they do not give rise to the same issues with unaware text modifications.