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. Whether 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.

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 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, as can be implemented between JIS and Shift-JIS.

Multistage Tables

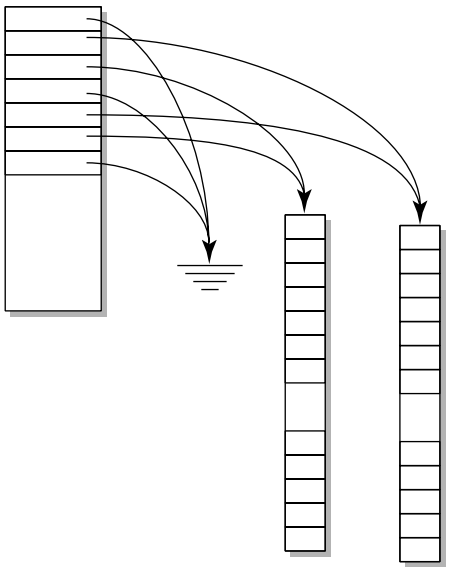
Tables require space. Even small character sets often map to characters from several different blocks in the Unicode Standard, and thus may contain up to 64K entries in at least one direction. Several techniques exist to reduce the memory space requirements for mapping tables. Such techniques apply not only to transcoding tables, but also to many other tables needed to implement the Unicode Standard, including character property data, 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 64K-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 penalties 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 (high-byte) tables are a commonly employed mechanism to reduce table size (see *Figure 5-1*). They use an array of 256 pointers and a default value. If a pointer is `NULL`, the returned value is the default. Otherwise, the pointer references a block of 256 values.

Figure 5-1. Two-Stage Tables



Optimized Two-Stage Table. Wherever any blocks are identical, the pointers just point to the same block. For transcoding tables, this case occurs generally for a block containing only mappings to the "default" or "unmappable" character. Instead of using `NULL` pointers and a default value, one "shared" block of 256 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.

Given an arbitrary 64K table, it is a simple matter to write a small utility that can calculate the optimal number of stages and their width.

7-Bit or 8-Bit Transmission

Some transmission protocols use ASCII control codes for flow control. Others, including some UNIX mailers, are notorious for their restrictions to 7-bit ASCII. In these cases, transmissions of Unicode-encoded text must be encapsulated. A number of encapsulation protocols exist today, such as `uuencode` and `BinHex`. These protocols can be combined with compression in the same pass, thereby reducing the transmission overhead.

The UTF-8 Transformation Format described in *Section 2.3, Encoding Forms*, and in *Section 3.8, Transformations*, may be used to transmit Unicode-encoded data through 8-bit transmission paths.

The UTF-7 Transformation Format, which is defined in RFC-2152, also exists for use with MIME.

Mapping Table Resources

To assist and guide implementers, the Unicode Standard provides a series of mapping tables on the accompanying CD-ROM. Each table consists of one-to-one mappings from the Unicode Standard to another published character standard. The tables include occasional multiple mappings. Their primary function is to identify the characters in these standards in the context of the Unicode Standard. In many cases, data conversion between the Unicode Standard and other standards will be application-dependent or context-sensitive. Many vendors maintain mapping tables for their own character standards.

> ***Disclaimer***
>
> The content of all mapping tables has been verified as far as possible by the Unicode Consortium. However, the Unicode Consortium does not guarantee that the tables are correct in every detail. The mapping tables are provided for informational purposes only. The Unicode Consortium is not responsible for errors that may occur either in the mapping tables printed in this volume or found on the CD-ROM, or in software that implements those tables. All implementers should check the relevant international, national, and vendor standards in cases where ambiguity of interpretation may occur.

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, with a width of 16 bits, fulfills the requirement.

The width of `wchar_t` is compiler-specific and can be as little 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 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. 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 transiently use 32-bit quantities to represent Unicode characters during processing. The internal workings of this representation are treated as a black box and are not Unicode-conformant. In particular, any API or runtime library interfaces that accept strings of 32-bit characters are not Unicode-conformant. If such an implementation interchanges 16-bit Unicode characters with the outside world, then this interchange can be conformant as long as the interface for this interchange complies with the requirements of *Chapter 3, Conformance*.

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.

Unassigned and Private Use Character Codes

There are two classes of character code values that even a "complete" implementation of the Unicode Standard cannot necessarily interpret correctly:

- Character code values that are unassigned
- Character code values in the Private Use Area for which no private agreement exists

An implementation should not attempt to interpret such code values. Options for rendering such unknown code values include printing the character code value as four hexadecimal digits, printing a black or white box, using appropriate glyphs such as ▨ for unassigned and ▯ for private use, or simply displaying nothing. In no case should an implementation *assume* anything else about the character's properties, nor should it blindly delete such characters. It should not unintentionally transform them into something else.

Interpretable but Unrenderable Characters

An implementation may receive a character that is an assigned 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 it in a default manner. An implementation can distinguish between unrenderable (but assigned) characters and unassigned code values by printing the former with distinctive glyphs that give some general indication of their type, such as [L], [Я], [Θ], [ָ], [か], [ב], [ش], [क], [ש], [가], [漢], and so on.

Reassigned Characters

Some character code values, primarily Hangul, were assigned in versions of the Unicode Standard earlier than 2.0, but have been reassigned because the characters were moved (see the Transcoding Tables on the CD-ROM). Such code values should be recognized and converted into the correct Version 3.0 character code values where possible. In some cases, a Version 3.0 application may still need to emit Version 1.1 character codes to communicate with some Version 1.1 applications. This issue is limited to particular early Unicode applications and data sets and is not a general problem. The Unicode Consortium has the policy to never reassign characters in the future.

5.4 Handling Surrogate Pairs

Surrogate pairs provide a mechanism for encoding 917,476 characters without requiring the use of 32-bit characters. Because predominantly infrequently used characters will be assigned to surrogate pairs, not all implementations need to handle these pairs initially. It is widely expected that surrogate pairs will be assigned in the not too distant future.

High-surrogates and low-surrogates are assigned to disjoint ranges of code positions. Nonsurrogate characters can never be assigned to these ranges. Because the high- and low-surrogate ranges are disjoint, determining character boundaries requires at most scanning one preceding or following Unicode code value, without regard to any other context. This approach enables efficient random access, which is not possible with encodings such as Shift-JIS.

In well-formed text, a low-surrogate can be preceded only by a high-surrogate and not by a low-surrogate or nonsurrogate. A high-surrogate can be followed only by a low-surrogate and not by a high-surrogate or nonsurrogate.

Surrogates are also designed to work well with implementations that do not recognize them. For example, the valid sequence of Unicode characters [0048] [0069] [0020] [D800] [DC00] [0021] [0021] would be interpreted by a Version 1.0-conformant implementation as “Hi <*unrecognized*><*unrecognized*>!!” This outcome is only slightly worse than that produced by a Version 3.0-conformant implementation that did not support that particular surrogate pair, and so interpreted the sequence as “Hi <*unrecognized*>!!”

As long as an implementation does not remove either surrogate or insert another character between them, the data integrity is maintained. Moreover, even if the data become corrupted, the data corruption is localized, unlike with some multibyte encodings such as Shift-JIS or EUC. Corrupting a single Unicode value affects only a single character. Because the high- and low-surrogates are disjoint and always occur in pairs, errors are prevented from propagating through the rest of the text.

Implementations can have different levels of support for surrogates, based on two primary issues:

- Does the implementation interpret a surrogate pair as the assigned single character?
- Does the implementation guarantee the integrity of a surrogate pair?

The decisions on these issues give rise to three reasonable levels of support for surrogates as shown in *Table 5-1*.

Table 5-1. Surrogate Support Levels

Support Level	Interpretation	Integrity of Pairs
None	No pairs	Does not guarantee
Weak	Non-null subset of pairs	Does not guarantee
Strong	Non-null subset of pairs	Guarantees

Example. The following sentence could be displayed in three different ways, assuming that both the weak and strong implementations have Phoenician fonts but no hieroglyphics: "The Greek letter α corresponds to <*hieroglyphic-high*><*hieroglyphic-low*> and to <*Phoenician-high*><*Phoenician-low*>." The ■ in *Table 5-2* represents any visual representation of an uninterpretable single character by the implementation.

Table 5-2. Surrogate Level Examples

None	"The Greek letter α corresponds to ■ ■ and to ■ ■."
Weak	"The Greek letter α corresponds to ■ ■ and to <*Phoenician*>."
Strong	"The Greek letter α corresponds to ■ and to <*Phoenician*>."

Many implementations that handle advanced features of the Unicode Standard can easily be modified to support a weak surrogate implementation. 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 values with a single keypress, much as an Arabic keyboard can have a "*lam-alef*" key that generates a sequence of two characters, *lam* and *alef*.
- Character display and measurement can be handled by treating specific surrogate pairs as ligatures, in the same way as "f" and "i" are joined to form the single glyph "fi".
- Truncation can be handled with the same mechanism as used to keep combining marks with base characters. (For more information, see *Section 5.15, Locating Text Element 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 Unicode code values 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.

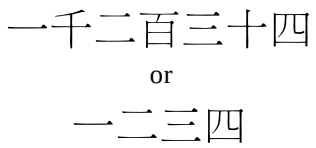
When converting binary numerical values to a visual form, digits can be chosen from different scripts. For example, the value *twenty* can be represented either by U+0032 DIGIT TWO, U+0030 DIGIT ZERO or by U+0968 DEVANAGARI DIGIT TWO, U+0966 DEVANAGARI DIGIT ZERO, or by U+0662 ARABIC-INDIC DIGIT TWO, 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 tables providing 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



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 5.16, 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 Handling Properties

The Unicode Standard provides detailed information on character properties (see *Chapter 4, Character Properties*, and the Unicode Character Database on the accompanying CD-ROM). These properties can be used by implementers to implement a variety of low-level processes. Fully language-aware and higher-level processes will need additional information.

A two-stage table, as described in *Section 5.1, Transcoding to Other Standards*, can also be used to handle mapping to character properties or other information indexed by character code. For example, the data from the Unicode Character Database on the accompanying CD-ROM can be represented in memory very efficiently as a set of two-stage tables.

Individual properties are common to large sets of characters and therefore lend themselves to implementations using the shared blocks.

Many popular implementations are influenced by the POSIX model, which provides functions for separate properties, such as `isalpha`, `isdigit`, and so on. Implementers of Unicode-based systems and internationalization libraries need to take care to extend these concepts to the full set of Unicode characters correctly.

In Unicode-encoded text, combining characters participate fully. In addition to providing callers with information about which characters have the combining property, implementers and writers of language standards need to provide for the fact that combining characters assume the property of the preceding base character (see also *Section 3.5, Combination*, and *Section 5.16, Identifiers*). Other important properties, such as sort weights, may also depend on a character's context.

Because the Unicode Standard provides such a rich set of properties, implementers will find it useful to allow access to several properties at a time, possibly returning a string of bit-fields, one bit-field per character in the input string.

In the past, many existing standards, such as the C language standard, assumed very minimalist "portable character sets" and geared their functions to operations on such sets. As the Unicode encoding itself is increasingly becoming *the* portable character set, implementers are advised to distinguish between historical limitations and true requirements when implementing specifications for particular text processes.

5.7 Normalization

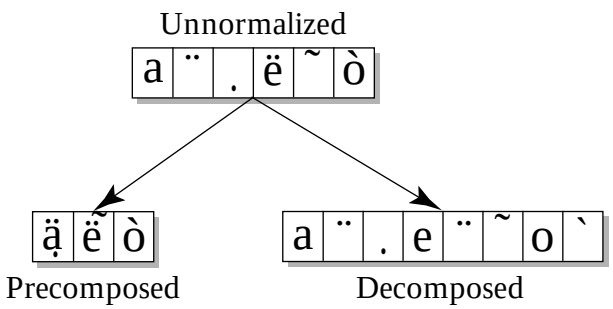
Alternative Spellings. The Unicode Standard contains explicit codes for the most frequently used accented characters. These characters can also be composed; in the case of accented letters, characters can be composed from a base character and nonspacing mark(s).

The Unicode Standard provides a table of normative spellings (decompositions) of characters that can be composed using a base character plus one or more nonspacing marks. These tables can be used to unify spelling in a standard manner in accordance with *Section 3.10, Canonical Ordering Behavior*. 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. Changes may occur as the result of character additions in the future. When new precomposed characters are added, mappings between those characters and corresponding composed character sequences will be added. Similarly, if a new combining mark is added to this standard, it may allow decompositions for precomposed characters that did not have decompositions before.

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 5.14, 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 and for conformance to those forms are defined in Unicode Technical Report #15, "Unicode Normalization Forms," on the CD-ROM or the up-to-date version on the Unicode Web site. For further information see *Chapter 3, Conformance; Chapter 4, Character Properties;* and *Section 2.6, Combining Characters*.

5.8 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 Report #6, "A Standard Compression Scheme for Unicode," on the CD-ROM or the up-to-date version on the Unicode Web site.

Encoding forms defined in *Section 2.3, 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 Unicode-encoded text.

5.9 Line Handling

Newlines are represented on different platforms by CR, LF, NL, or CRLF. Not only are newlines represented by different characters on different platforms, but they also have ambiguous behavior even on the same platform. With the advent of the Web, where text on a single machine can arise from many sources, this inconsistency causes a significant problem.

Unfortunately, these characters are often transcoded directly into the corresponding Unicode code values when a character set is transcoded. For this reason, even programs handling pure Unicode text must deal with the problems.

For detailed guidelines, see Unicode Technical Report #13, "Unicode Newline Guidelines," on the CD-ROM or the up-to-date version on the Unicode Web site.

5.10 Regular Expressions

Byte-oriented regular expression engines require extensions to successfully handle Unicode. 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," on the CD-ROM or the up-to-date version on the Unicode Web site.

5.11 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 Report #10, "Unicode Collation Algorithm," on the CD-ROM or the up-to-date version on the Unicode Web site 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 Östersjö 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. 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.

For interchange purposes, it is becoming common to use tagged information, which *is* embedded in the text. Unicode Technical Report #7, "Plane 14 Characters for Language Tags," which is found on the CD-ROM or in its up-to-date version on the Unicode Web site, provides a proposed mechanism for representing language tags. Like most tagging mechanisms, these language tags are stateful: a start tag establishes an attribute for the text, and an end tag concludes it.

Working with Language Tags

Avoiding Language Tags. Because of the extra implementation burden, language tags should be avoided in plain text unless language information is required and it is known that the receivers of the text will properly recognize and maintain the tags. However, where

language tags must be used, implementers should consider the following implementation issues involved in supporting language information with tags and decide how to handle tags where they are *not* fully supported. This discussion applies to any mechanism for providing language tags in a plain text environment.

Display. Language tags themselves are not displayed. This choice may not require modification of the displaying program, if the fonts on that platform have the language tag characters mapped to zero-width, invisible glyphs. Language tags may need to be taken into account for special processing, such as hyphenation or choice of font.

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

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

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

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

Higher-Level Protocols. Higher-level protocols such as HTML or MIME may also supply language tags. Language tags should be avoided wherever higher-level protocols, such as a rich-text format, provide language information. Not only does this approach avoid problems, but it also avoids cases where the higher-level protocol and the language tags disagree.

Language Tags and Han Unification

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.

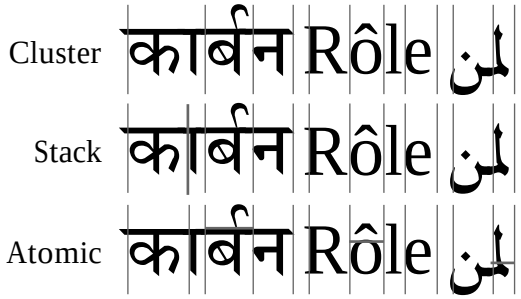
It is common even in printed matter to render passages of foreign language text in native-language fonts, just for familiarity. For example, Chinese text in a Japanese document can commonly be rendered in a Japanese font.

5.12 Editing and Selection

Consistent Text Elements

A user interface for editing is most intuitive when the text elements are consistent (see *Figure 5-4*). In particular, the editing actions of deletion, selection, mouse-clicking, and cursor-key movement should act as though they have a consistent set of boundaries. For example, hitting a leftward-arrow should result in the same cursor location as delete. *This synchronization gives a consistent, single model for editing characters.*

Figure 5-4. Consistent Character Boundaries



Three types of boundaries are generally useful in editing and selecting within words.

Cluster Boundaries. Cluster boundaries occur in scripts such as Devanagari. Selection or deletion using cluster boundaries means that an entire cluster (such as *ka + vowel sign a*) or a composed character (*o + circumflex*) is selected or deleted as a single unit.

Stacked Boundaries. Stacked boundaries are generally somewhat finer than cluster boundaries. Free-standing elements (such as *vowel sign a*) can be independently selected and deleted, but any elements that "stack" (such as *o + circumflex*, or vertical ligatures such as Arabic *lam + meem*) can be selected only as a single unit. Stacked boundaries treat all composed character sequences as single entities, much like precomposed characters.

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 and one that is currently in use on the Macintosh and some other systems. 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 are 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.

5.13 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 DTP (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 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, monospaced 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 the large majority of applications. They can, however, also supply more sophisticated truncation routines. (See also *Section 5.14, 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 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.

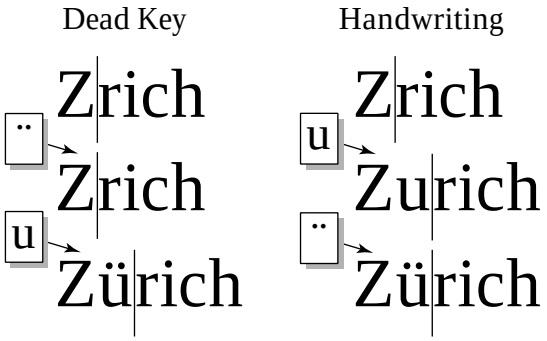
Line-break algorithms generally use state machines for determining word breaks. Such algorithms can be easily adapted to prevent separation of nonspacing marks from base characters. (See also the discussion in *Section 5.17, Sorting and Searching; Section 5.7, Normalization;* and *Section 5.15, Locating 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 working 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*).

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.

Figure 5-5. Dead Keys Versus Handwriting Sequence



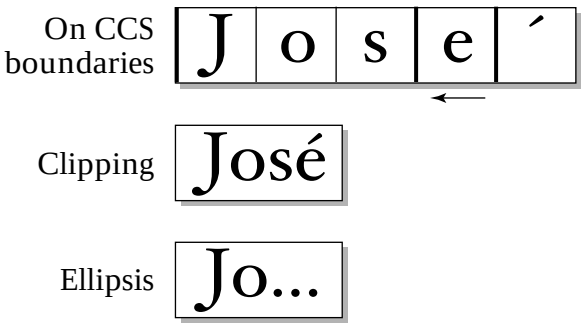
Truncation

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 performing 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 value (see *Figure 5-6*). (See *Section 5.15, Locating Text Element 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.

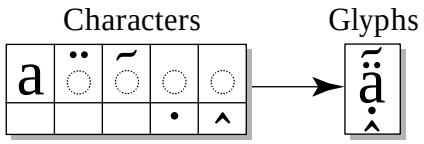
A different option is simply to clip the characters graphically. However, the result may look ugly. Also, if the clipping occurs between characters, it may not give any visual feedback that characters are being omitted. A graphic or ellipsis can be used to give this visual feedback.

5.14 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 in these fonts, especially with international characters.

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 2.6, 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 14, Code Charts.*)

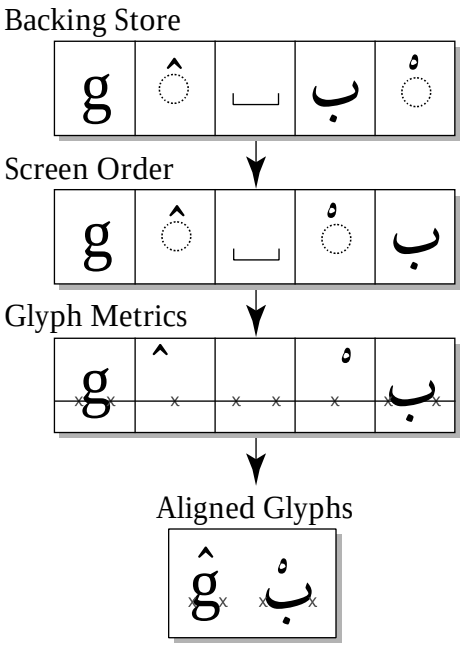
Figure 5-8. Fallback Rendering

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 3.5, Combination*). Defective combining character sequences should be rendered as if they had a space as a base character.

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 *Section 3.12, Bidirectional Behavior*.)

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 separate from their base characters.

Figure 5-10. Justification

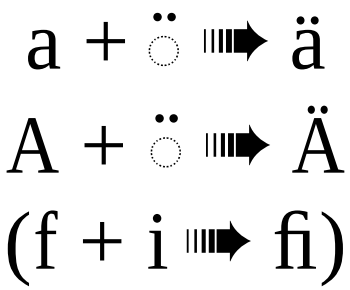
Because nonspacing marks always follow their base character, proper justification adds letterspace between characters only if the second character is a base character.

Positioning Methods

A number of different 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

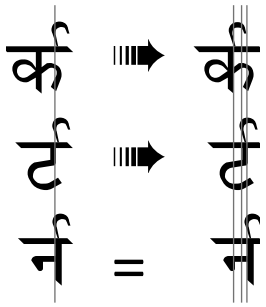


Positioning with ligatures is perhaps the simplest method of supporting nonspacing marks. Whenever there is a small, fixed set, such as those corresponding to the precomposed characters of 8859-1 (Latin1), this method is straightforward to apply. Because the composed character sequence almost always has the same width as the base character, rendering, measurement, and editing of these characters are much easier than for the general case of ligatures.

If a 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 upon their general shape and width. According to the class of the base glyph, a particular glyph is chosen for a nonspacing mark.

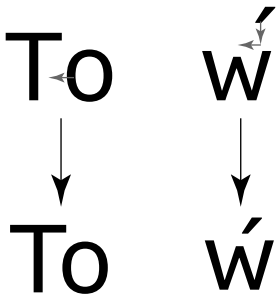
Figure 5-12. Positioning with Contextual Forms



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 arbitrarily positioned.

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.

5.15 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 unambiguously decide boundaries. 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. The determination of those boundaries is often critical to the performance of general software, so it is important to be able to make such a determination as quickly as possible.

The following boundary determination mechanism provides a straightforward and efficient way to determine word boundaries. It builds upon the uniform character representation of the Unicode Standard, while handling the large number of characters and special features such as combining marks and surrogates in an effective manner. As this boundary determination mechanism lends itself to a completely data-driven implementation, it can be customized to particular locales according to special language or user requirements without recoding. In particular, word, line, and sentence boundaries will need to be customized according to locale and user preference. In Korean, for example, lines may be broken either at spaces (as in Latin text) or on ideograph boundaries (as in Chinese).

For some languages, this simple method is not sufficient. For example, Thai linebreaking requires the use of dictionary lookup, analogous to English hyphenation. An implementation therefore may need to provide means to override or subclass the standard, fast mechanism described in the “Boundary Specification” subsection in this section.

The large character set of the Unicode Standard and its representational power place requirements on both the specification of text element boundaries and the underlying implementation. The specification needs to allow for the designation of large sets of characters sharing the same characteristics (for example, uppercase letters), while the implementation must provide quick access and matches to those large sets.

The mechanism also must handle special features of the Unicode Standard, such as combining or nonspacing marks, conjoining jamo, and surrogate characters.

The following discussion looks at two aspects of text element boundaries: the specification and the underlying implementation. Specification means a way for programmers and localizers to specify programmatically where boundaries can occur.

Boundary Specification

A boundary specification defines different classes, then lists the rules for boundaries in terms of those classes. Remember that characters may be represented by a sequence of two surrogates. The character classes are specified as a list, where each element of the list is

- A literal character
- A range of characters
- A property of a Unicode character, as defined in the Unicode Character Database

- A removal of the following elements from the list

For the formal description of the notation used in this section, see *Section 0.2, Notational Conventions*. Additional notational conventions used in this section are as follows:

÷ Allow break here.

× Do not allow break here.

An *underscore* ("_") is used to indicate a space in examples.

There is always a boundary at the beginning and at the end of a string of text. As with typical regular expressions, the longest match possible is used. For example, in the following, the boundary is placed after as many Y's as possible:

X Y ÷ ¬ ***X***

(In this book, the rules are numbered for reference.) Some additional constraints are reflected in the specification. These constraints make the implementation significantly simpler and more efficient and have not been found to be limitations for natural language use.

1. ***Limited context.*** Given boundaries at positions X and Y, then the positions of any other boundaries between X and Y do not depend on characters outside of X and Y (so long as X and Y remain boundary positions).

   For example, with boundaries at "ab÷cde", changing "a" to "A" cannot introduce a new boundary, such as at "Ab÷cd÷e".

2. ***Single boundaries.*** Each rule has exactly one boundary position. Because of rule (1), this restriction is more a limitation on the specification methods, because a rule with two boundaries could generally be expressed as two rules.

   For example, "ab÷cd÷ef" could be broken into "ab÷cd" and "cd÷ef".

3. ***No conflicts.*** Two rules cannot have initial portions that match the same text, but with different boundary positions.

   For example, "x÷abc" and "a÷bc" cannot be part of the same boundary specification.

4. ***No overlapping sets.*** For efficiency, two character sets in a specification cannot intersect. A later character set definition will *override* a previous one, removing its characters from the previous set.

   For example, the second set specification removes "AEIOUaeiou" from the first:

   Let = [Lu][Ll][Lt][Lm][Lo]

   EngVowel = AEIOUaeiou

5. ***No more than 256 sets.*** This restriction is purely an implementation detail to save on storage.

6. ***Ignore degenerates.*** Implementations need not make special provisions to get marginally better behavior for degenerate cases that never occur in practice, such as an *A* followed by an Indic combining mark.

These rules can also be approximated with pair tables, where only one character before, and one character after, a possible break point are considered. Although this approach does not generally give the correct results, it is often sufficient if languages are restricted. However, the grapheme boundaries are designed to be fully expressed using pair tables.

Example Specifications

Different issues are present with different types of boundaries, as the following discussion and examples should make clear. In this section, the rules are somewhat simplified, and not all edge cases are included. In particular, characters such as control characters and format characters do not cause breaks. Permitting such cases would complicate each of the examples (and is left as an exercise for the reader). In addition, it is intended that the rules themselves be localizable; the examples provided here are not valid for all locales.

Rather than listing the contents of the character sets in these examples, the contents are simply explained.

If an implementation uses a state table, the performance does not depend on the complexity or number of rules. The only feature that does affect performance is the number of characters that may match *after* the boundary position in a rule that is matched.

Grapheme Boundaries

A *grapheme* is a minimal unit of a writing system, just as a phoneme is a minimal unit of a sound system. In some instances, a grapheme is represented by more than one Unicode character, such as in Indic languages. As far as a user is concerned, the underlying representation of text is not important, but it is paramount that an editing interface present a uniform implementation of what the user thinks of as graphemes. Graphemes should behave as units in terms of mouse selection, arrow key movement, backspacing, and so on. For example, if an accented character is represented by a combining character sequence, then using the right arrow key should skip from the start of the base character to the end of the last combining character. This situation is analogous to a system using conjoining jamo to represent Hangul syllables, where they are treated as single graphemes for editing. In those circumstances where end users need character counts (which is actually rather rare), the counts need to correspond to the users' perception of what constitutes a grapheme.

The principal requirements for general character boundaries are the handling of combining marks, Hangul conjoining jamo, and Indic and Tibetan character clusters. See *Table 5-3*.

Table 5-3. Grapheme Boundaries

Character Classes	
CR	Carriage Return
LF	Line Feed
Format	All other control or format characters
Virama	Indic viramas
Joining	All combining characters, plus Tibetan subjoined characters
L	Hangul leading jamo
V	Hangul vowel jamo
T	Hangul trailing jamo
Lo	Other letters
Other	All other characters

Table 5-3. Grapheme Boundaries (Continued)

Rules			
Always break before an LF, unless it is preceded by a CR. Always break before and after all other format or control characters.			
¬ ***CR***	÷	***LF***	**(1)**
CR	÷	¬ ***LF***	**(2)**
	÷	**(*CR* | Format)**	**(3)**
(Format | *LF*)	÷		**(4)**
Break before viramas and joining characters only if they are preceded by controls or formats.			
(Format | *CR* | *LF*)	÷	**(Virama | Joining)**	**(5)**
Break before and after Hangul jamo unless they are in an allowable sequence.			
¬ ***L***	÷	***L***	**(6)**
¬ **(*L* | *V*)**	÷	***V***	**(7)**
¬ **(*L* | *V* | *T*)**	÷	***T***	**(8)**
(*L* | *V* | *T*)	÷	¬ **(*L* | *V* | *T*)**	**(9)**
Don't break between viramas and other letters. This rule provides for Indic graphemes, where virama will link character clusters together.			
Virama	×	***Lo***	**(10)**
Break before any other characters.			
	÷	**Other**	**(11)**

Word Boundaries

Word boundaries are used in a number of different contexts. The most familiar ones are double-click mouse selection, "move to next word," and detection of whole words for search and replace.

For the search and replace option of "find whole word," the rules are fairly clear. The boundaries are between letters and nonletters. Trailing spaces cannot be counted as part of a word, because searching for "abc" would then fail on "abc_".

For word selection, the rules are somewhat less clear. Some programs include trailing spaces, whereas others include neighboring punctuation. Where words do not include trailing spaces, sometimes programs treat the individual spaces as separate words; other times they treat an entire string of spaces as a single word. (The latter fits better with usage in search and replace.)

- Word boundaries can also be used in so-called *intelligent cut and paste*. With this feature, if the user cuts a piece of text on word boundaries, adjacent spaces are collapsed to a single space. For example, cutting "quick" from "The_quick_fox" would leave "The_ _fox". Intelligent cut and paste collapses this text to "The_fox".

This discussion outlines the case where boundaries occur between letters and nonletters, and there are no boundaries between nonletters. It also includes Japanese words (for word selection). In Japanese, words are not delimited by spaces. Instead, a heuristic rule is used in which strings of either *kanji* (ideographs) or katakana characters (optionally followed by strings of hiragana characters) are considered words. See *Table 5-4*.

Table 5-4. Word Boundaries

Character Classes

CR	Carriage Return
LF	Line Feed
Sep	LS, PS
TAB	Tab character
Let	Letter
Com	Combining mark
Hira	Hiragana
Kata	Katakana
Han	Han ideograph (Kanji)

Rules

Always break before an LF, unless it is preceded by a CR. Always break before and after all other format or control characters, including TAB.

¬ *CR*	**÷**	***LF***	**(1)**
CR	**÷**	**¬ *LF***	**(2)**
	÷	**(*CR* | Sep | *TAB*)**	**(3)**
(Sep | *TAB* | *LF*)	**÷**		**(4)**

Break between letters and nonletters. Include trailing nonspacing marks as part of a letter.

¬ (Let | Com)	**÷**	**Let**	**(5)**
Let Com*	**÷**	**¬ Let**	**(6)**

Handle Japanese specially for word selection. Treat clusters of kanji or katakana (with or without following hiragana) as single words. Break when preceded by other characters (such as punctuation). Include nonspacing marks.

¬ (Hira | Kata | Han | Com)	**÷**	**Hira | Kata | Han**	**(7)**
Hira Com*	**÷**	**¬ Hira**	**(8)**
Kata Com*	**÷**	**¬ (Hira | Kata)**	**(9)**
Han Com*	**÷**	**¬ (Hira | Han)**	**(10)**

- One could also generally break between any letters of different scripts. In practice—except for languages that do not use spaces—this possibility is a degenerate case.

Line Boundaries

Line boundaries determine acceptable locations for line-wrap to occur without hyphenation. (More sophisticated line-wrap also makes use of hyphenation, but generally only in cases where the natural line-wrap yields inadequate results.) Note that this approach is very similar to word boundaries but *not* generally identical.

For the purposes of line-break, a composed character sequence should generally be treated as though it had the same properties as the base character. The nonspacing marks should not be separated from the base character.

Nonspacing marks may be exhibited in isolation—that is, over a space or nonbreaking space. In that case, the entire composed character sequence is treated as a unit. If the composed character sequence consists of a *no-break space* followed by nonspacing marks, then it does not generally allow line-breaks before or after the sequence. If the composed character sequence consists of any other space followed by nonspacing marks, then it generally does allow line-breaks before or after the sequence.

There is a degenerate case where a nonspacing mark occurs as the first character in the text or after a line or paragraph separator. In that case, the most consistent treatment for the line-break is to treat the nonspacing mark as though it were applied to a space. See *Table 5-5*.

Table 5-5. Line Boundaries

Character Classes	
CR	Carriage Return
LF	Line Feed
ZWSP	Zero-width space
ZWNBSP	Zero-width no-break space
Sp	Spaces
Break	Mandatory break (for example, Paragraph Separator)
Com	All combining characters (including Tibetan subjoined characters), plus medial and final conjoining Hangul jamo
Ideographic	Ideographic characters
Alphabetic	Alphabetic characters and most symbols
Exclam	Terminating characters like exclamation point
Syntax	Solidus (‘/’)
Open	Open punctuation, such as ‘(’
Close	Closing punctuations, such as ‘)’
Quote	Ambiguous quotations marks, such as " and '
NonStarter	Small hiragana and katakana characters
HyphenMinus	U+002D
Insep	Ellipsis characters and leaders
Number	Digits
NumericPrefix	Characters such as ‘$’
NumericPostfix	Characters such as ‘%’
NumericPrefix	Characters such as ‘$’
NumericPostfix	Characters such as ‘%’
NumericInfix	Characters such as period and comma that occur within European numbers

Table 5-5. Line Boundaries (Continued)

Base	Base characters
NonBase	Nonbase characters (control characters and format characters)
All	All Unicode characters

Note: For a precise specification of these classes, see Unicode Technical Report #14, "Line Breaking Properties," on the CD-ROM or the up-to-date version on the Unicode Web site. (The classes have slightly different names here for consistency.)

Rules

Explicit breaks and nonbreaks:

- Always break after hard line-breaks (but never between CR and LF). There is a break opportunity after every ZWSP, but not a hard break.
- Do not break before spaces or hard line-breaks.
- Do not break before or after ZWNBSP.

CR	×	***LF***	**(1)**
(*CR* | *LF* | Break | *ZWSP*)	÷		**(2)**
	×	**(*SP* | *CR* | *LF* | Break | *ZWSP* | *ZWNBSP*)**	**(3)**
ZWNBSP	×		**(4)**

Combining marks:

- Do not break graphemes (before combining marks). Virama and jamos are merged with the proper classes so they work correctly.

In all of the following rules:

- If a space is the base character for a combining mark, the space is changed to type AL.
- At any possible break opportunity between Com and a following character, Com behaves as if it had the type of its base character. If there is no base, the Com behaves like AL.

	×	**Com**	**(5)**
Sp Com	→	**Alphabetic Com**	**(6)**
Base Com	→	**Base Base**	**(7)**
NonBase Com	→	**NonBase Alphabetic**	**(8)**

Handle opening and closing:

These have special behavior with respect to spaces.

- Do not break before exclamations, syntax characters, or closing punctuation, even after spaces.

Table 5-5. Line Boundaries (Continued)

- Do not break after opening punctuations, even after spaces.
- Do not break within quotations and opening punctuations, or between closing punctuation and ideographs, even with intervening spaces.

	×	**(Exclam | Syntax | Close)**	**(9)**
Open	×		**(10)**
Quote	×	**Open**	**(11)**
Close	×	**Ideographic**	**(12)**

Once opening and closing are handled, spaces can be finished up.

- Break after spaces

Sp	÷		**(13)**

Special case rules:

- Do not break before or after ambiguous quotation marks.
- Do not break before no-starts or HyphenMinus.
- Do not break between two ellipses, or between letters or numbers and ellipsis, such as in ‘9...’, ‘a...’, ‘H...’.
- Do not break within alphabetics and numbers, or numbers and alphabetics, or ideographs and numeric suffixes.

	×	**(Quote | NonStarter | HyphenMinus)**	**(14)**
Quote	×		**(15)**
(Insep | Number | Ideographic | Alphabetic)	×	**Insep**	**(16)**
Alphabetic	×	**Number**	**(17)**
Number	×	**Alphabetic**	**(18)**
Ideographic	×	**NumericPostfix**	**(19)**

Numbers are of various forms that should not be broken across lines—for example, $(12.35), 12, (12)¢, 12.54¢, and so on.

These are approximated with the following rule. (Some cases were already handled, such as ‘9,’ and ‘[9’.)

NumericPrefix	×	**(Number | Open | HyphenMinus)**	**(20)**
(HyphenMinus | Syntax | Number | NumericInfix)	×	**Number**	**(21)**
Number	×	**NumericPostfix**	**(22)**
Close	×	**NumericPostfix**	**(23)**

Table 5-5. Line Boundaries (Continued)

Join alphabetic letters and break everywhere else.

Alphabetic	×	**Alphabetic**	**(24)**
	÷	**All**	**(25)**
All	÷		**(26)**

Sentence Boundaries

Sentence boundaries are often used for triple-click or some other method of selecting or iterating through blocks of text that are larger than single words.

Plain text provides inadequate information for determining good sentence boundaries. Periods, for example, can either signal the end of a sentence, indicate abbreviations, or be used for decimal points. Without analyzing the text semantically, it is impossible to be certain which of these usages is intended (and sometimes ambiguities still remain). See *Table 5-6*.

Table 5-6. Sentence Boundaries

Character Classes

CR	Carriage Return
LF	Line Feed
Sep	LS, PS
Sp	Space separator
Term	!?
Dot	Period
Cap	Uppercase, titlecase, and noncased letters
Lower	Lowercase
Open	Open punctuation
Close	Close punctuation, period, comma, ...

Rules

Always break before an LF, unless it is preceded by a CR. Always break before and after separating control or format characters.

¬ *CR*	÷	***LF***	**(1)**
CR	÷	**¬ *LF***	**(2)**
	÷	**(*CR* | Sep)**	**(3)**
(Sep | *LF*)	÷		**(4)**

Break after sentence terminators, but include nonspacing marks, closing punctuation, trailing spaces, and (optionally) a paragraph separator.

Term Close* Sp* {Sep}	÷		**(5)**

Table 5-6. Sentence Boundaries (Continued)

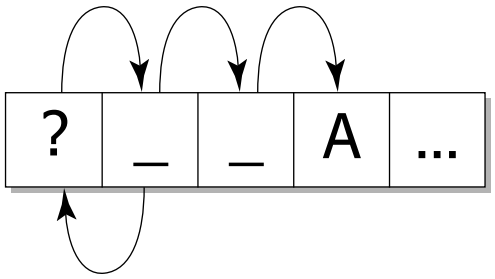
Handle a period specially, as it may be an abbreviation or numeric period—and not the end of a sentence. Don't break if it is followed by a lowercase letter instead of an uppercase letter. **Dot Close* Sp+ ÷ Open* ¬ Lower (6)**

Random Access

A further complication is introduced by random access (see *Figure 5-14*). When iterating through a string from beginning to end, the preceding approach works well. It guarantees a limited context, and it allows a fresh start at each boundary to find the next boundary. By constructing a state table for the reverse direction from the same specification of the rules, reverse searches are possible. However, suppose that the user wants to iterate starting at a random point in the text. If the starting point does not provide enough context to allow the correct set of rules to be applied, then one could fail to find a valid boundary point. For example, suppose a user clicked after the first space in "?_ _A". On a forward iteration searching for a sentence boundary, one would fail to find the boundary before the "A", because the "?" hadn't been seen yet.

A second set of rules to determine a "safe" starting point provides a solution. Iterate backward with this second set of rules until a safe starting point is located, then iterate forward from there. Iterate forward to find boundaries that were located between the starting point and the safe point; discard these. The desired boundary is the first one that is not less than the starting point.

Figure 5-14. Random Access



This process would represent a significant performance cost if it had to be performed on every search. However, this functionality could be wrapped up in an iterator object, which preserves the information regarding whether it currently is at a valid boundary point. Only if it is reset to an arbitrary location in the text is this extra backup processing performed.

5.16 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 for the definition of identifier syntax.

Note that the task of parsing for identifiers and the task of locating word boundaries are related, and it is a straightforward procedure to restate the sample syntax provided here in

the form just discussed for locating text element boundaries. In this section, a more traditional BNF-style syntax is presented to facilitate incorporation into existing standards.

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.

> *These rules are no more complex than current rules in the common programming languages, except that they include more characters of different types.*

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

1. Incorporation of proper handling of combining marks
2. 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 CAPITAL LATIN LETTER A) are themselves not valid constituents of 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 will more likely *be* identical.

Specific Character Additions. Specific identifier syntaxes can be treated as slight modifications of the generic syntax based on character properties. Thus, 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.

For the notation used in this section, see *Section 0.2, 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*.

Implementations that require a stable definition of identifiers for conformance must reference a specific version of the Unicode Standard, such as Version 3.0.0, or explicitly list the particular subset of Unicode characters that fall under their definition of syntactic class.

Table 5-7. Syntactic Classes for Identifiers

Syntactic Class	Equivalent Category Set	Coverage
<identifier_start>	Lu, Ll, Lt, Lm, Lo, Nl	Uppercase letter, lowercase letter, titlecase letter, modifier letter, other letter, letter number
<identifier_extend>	Mn, Mc, Nd, Pc, Cf	Nonspacing mark, spacing combining mark, decimal number, connector punctuation, formatting code

5.17 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 always depend on the application and language, but for an implementation supporting the Unicode Standard, sorting and searching must also take into account the Unicode character equivalence and canonical ordering defined in *Chapter 3, Conformance*.

This section also discusses issues of adapting sublinear text searching algorithms, providing for fast text searching while still maintaining language-sensitivity, and using the same ordering algorithms that are used for collation. For more information, see Unicode Technical Report #10, "Unicode Collation Algorithm," on the CD-ROM or the up-to-date version on the Unicode Web site.

Culturally Expected Sorting

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 neither possible to arrange characters in an encoding in an order so that simple binary string comparison produces the desired collation order, nor is it possible to provide single-level sort-weight tables. The latter implies that character encoding details have only an indirect influence on culturally expected sorting.

To address the complexities of culturally expected sorting, a multilevel comparison algorithm is typically employed.[1] Each character in string is given several categories of *sort weights*. Categories can include alphabetic, case, and diacritic weights, among others.

1. A good example can be found in Denis Garneau, *Keys to Sort and Search for Culturally-Expected Results* (IBM document number GG24-3516, June 1, 1990), which addresses the problem for Western European languages, Arabic, and Hebrew.

In a first pass, these weights are accumulated into a *sort key* for the string. At the end of the first pass, the sort key contains a string of alphabetic weights, followed by a string of case weights, and so on. In a second pass, these substrings are compared by order of importance so that case and accent differences can either be fully ignored or applied only where needed to differentiate otherwise identical sort strings.

The first pass looks very similar to the decomposition of Unicode characters into base character and accent. The fact that the Unicode Standard permits multiple spellings (composed and composite) of the same accented letter turns out not to matter at all. If anything, a completely decomposed text stream can simplify the first implementation of sorting.

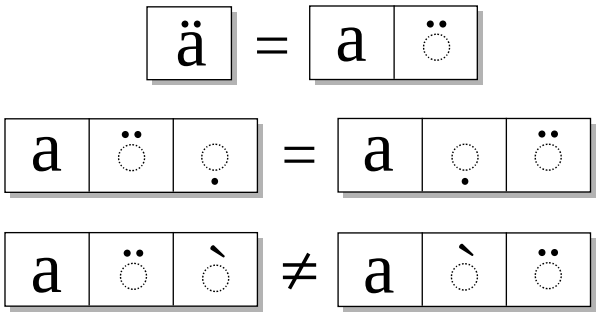
To provide a powerful, table-based approach to natural-language collation using Unicode characters, implementers need to consider providing full functionality for these features of language-sensitive algorithmic sorting:

- Four collation levels
- French or normal orientation
- Contracting or expanding characters
- Ordering of unmapped characters
- More than one level of ignorable characters

Unicode Character Equivalence

Section 3.6, Decomposition, and *Section 3.10, Canonical Ordering Behavior,* define equivalent sequences and provide an exact algorithm for determining when two sequences are equivalent. Equivalent sequences of Unicode characters should be collated in exactly the same way, no matter what the underlying storage is. *Figure 5-15* gives two examples of character equivalence.

Figure 5-15. Character Equivalence



Compatibility characters—especially where they have the same appearance—should also be collated in exactly the same way (for example, U+00C5 *Å* LATIN CAPITAL LETTER A WITH RING ABOVE and U+212B *Å* ANGSTROM SIGN).

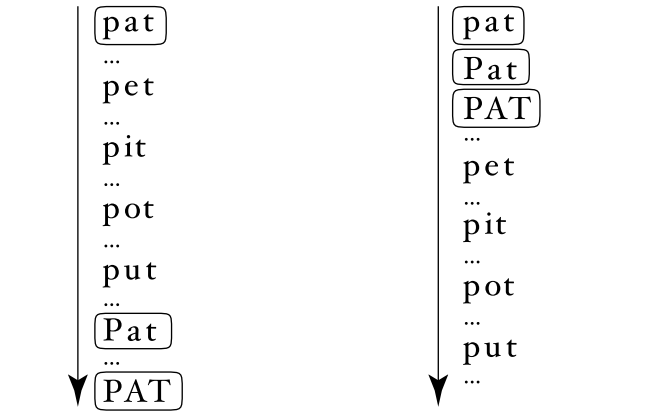
Similar Characters

Languages differ in what they consider similar characters, but users generally want characters that are similar (such as upper- and lowercase) to sort close to each other but not to be collated exactly the same way. If upper- and lowercase collated identically, words differing

only in case would appear in random order (see *Figure 5-16*). The same is true for accented characters.

Typically, there is an ordering among these similar characters. In English dictionaries, for example, lowercase precedes uppercase (the reverse of what happens with a naïve ASCII comparison). However, this ordering applies only when the strings are the same in all other respects. Otherwise, *Aachen* would sort after *azure*. Characters with accents often sort close to the base character, but different accents on the same base character always sort in a given order.

Figure 5-16. Naïve Comparison



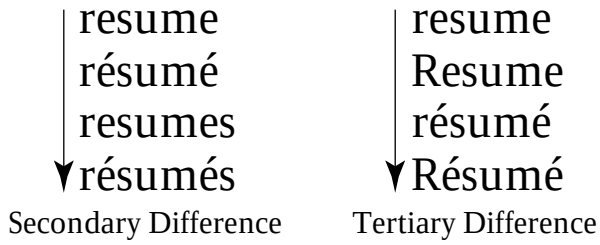
Levels of Comparison

The way to handle these problems is to use multiple levels of comparison and attach only a secondary or tertiary difference to the letters based on their case or accents (see *Figure 5-17*). Thus we get the following rules:

R1 ***Count secondary differences—only if there are no primary differences.***

R2 ***Count tertiary differences—only if there are no primary or secondary differences.***

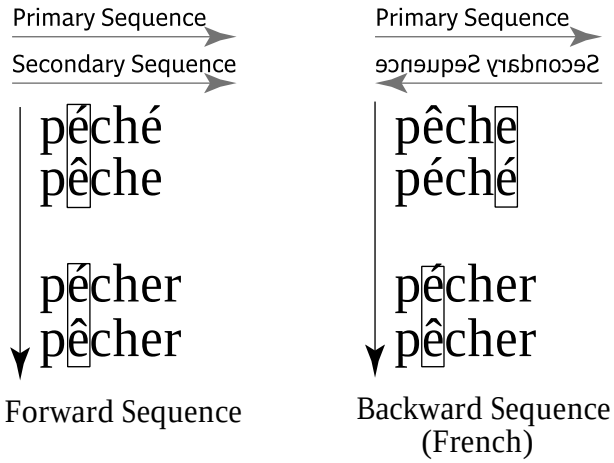
Figure 5-17. Levels of Comparison



In English and similar languages, accents make only a secondary difference, and case differences make only a tertiary difference. Ignorable characters are counted as secondary or tertiary differences. In other languages and scripts, other features map to secondary or tertiary differences.

French presents an interesting special case. In French sorting, the differences in accents later in the string are more important than those earlier in the string. In *Figure 5-18*, the two pairs of words are identical in the first five base characters. In English, the first accents are the most significant ones; in French, the first accents from the end are as the boxes show.

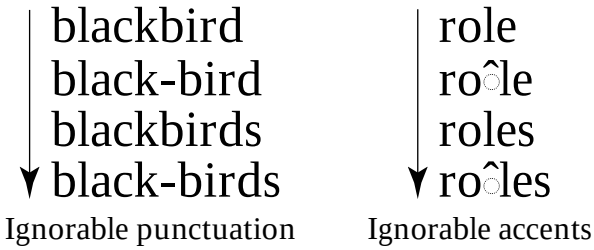
Figure 5-18. Orientation



Ignorable Characters

Another class of interesting cases involves ignorable characters (see *Figure 5-19*), such as spaces, hyphens, and some other punctuation. In this case, the character itself is ignored unless there are no stronger differences in the string.

Figure 5-19. Ignorable Characters



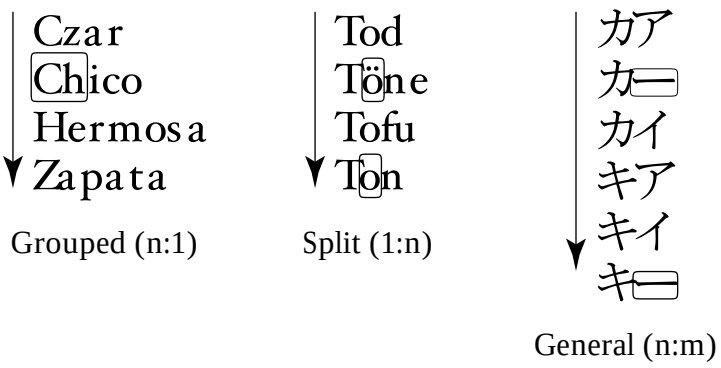
The general rule is as follows:

R3 Treat ignorable characters as having no primary difference.

Multiple Mappings

With many language collations such as traditional Spanish or German, one character may compare as if it were two, or two characters may compare as if they were one character (see *Figure 5-20*). In traditional Spanish orthography, for example, “Ch” sorts as a single letter—that is, after “Cz”; otherwise, it would come before “Ci”. In traditional German, “ö” sorts as if it were “oe”, putting it after “od” and before “of”.

Figure 5-20. Multiple Mappings



In Japanese, a ー *length mark* lengthens the vowel of the preceding character; depending on the vowel, the result will sort in a different order. For example, after the character カ "ka", the ー *length mark* indicates a long "a" and comes after ア "a"; after the character キ "ki", the ー *length mark* indicates a long "i" and comes after イ "i".

Look at the second character in each word in the third column of *Figure 5-20*. There are three different characters in the second position: ア "a", ー *length mark*, and イ "i". The general rule is as follows:

R4 N characters may compare as if they were M.

Collating Out-of-Scope Characters

Collation implements an ordering that matches the expectations of the user, based on rules of the user's language. Lists of terms encoded using the Unicode Standard may easily come from many different languages. These terms are all sorted according to the custom of the user's language.

For scripts and characters outside the use of a particular language, explicit rules may not exist. For example, Swedish and French have clear and different rules on sorting *ä* (either after *z* or as an accented character with a secondary difference from *a*), but neither defines a particular sorting order for the Han ideographs. Implementations supporting the Unicode Standard therefore typically provide a default ordering (like the culturally neutral ordering for ideographs used in this standard). Sorting for a Japanese user would still sort upper- and lowercase Latin letters in proximity. The *relative* ordering of scripts is typically configurable.

R5 Default to a common or culturally neutral ordering for out-of-scope characters.

Unmapped Characters

Another option is to treat out-of-scope characters as irrelevant. Such characters can include box forms, dingbats, and perhaps also alphabets that are not of concern for the user base of an implementation. Characters irrelevant to a collation sequence are usually not assigned weights; this choice saves space in the collation sequence. However, to provide a definitive sorting order, a position needs to be specified in the collation sequence for any unassigned character. For efficiency, collate any unassigned characters in Unicode bit order.

R6 Collate irrelevant characters in Unicode bit order, in a specified position.

Parameterization

For effective use of collation, programmers need to have a certain level of control and need to be able to get back sufficient information. One output needs to be the place in the string where a difference occurs, so that common initial substrings can be found. Notice that this position may be different in the two different strings. Because of multiple mappings, the first three characters in one string might be equivalent to the first four in the other.

Input

- Specify language rule
- Relative order of scripts
- Allows specification of precision

Output

- First point in each string where different
- Direction and precision of difference

Processing

- Can preprocess each string for fast comparison
- Process just as much as needed for a stand-alone comparison

Optimizations

Multiple-level comparison requires a bit more work than binary comparison. Although real-life studies put the overhead at around 50 percent, it often pays to first transform terms to be sorted into equivalent *sort keys*, which result in the same sorted list when subjected to a simple and fast binary comparison. (In the standard C library, the function `wcsxfrm` provides such a transformation.) These sort keys might consist of a string of base weights followed by strings for weights used for secondary and tertiary differences, as just discussed. Unlike Unicode code values, sort keys don't need to be 16-bit-based. Thus highly optimized functions, such as the `strcmp` function from the standard C library, can be used.

Sort keys can also be stored, obviating recomputation when a list needs to be re-sorted. Another straightforward optimization is to *compare as you go.* For each string, sort weights are assembled into sort keys only until a difference is located. This approach reduces the computation necessary when a difference is found early in the string.

Searching

Searching is subject to many of the same issues as comparison, such as a choice of a weak, strong, or exact match. 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 *Section 5.15, Locating Text Element 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 (see *Section 4.1, Case—Normative*, and *Section 7.1, Latin*). The preferred method of implementing case insensitivity uses the same mechanisms and tables described in the sorting discussions in the beginning of this section. In particular, it is advisable for many applications (for example, file systems) to treat a particular set of characters (i, I, ı *dotless-i,* İ *capital i with dot*) as a single equivalency class to guarantee reasonable results for Turkish.

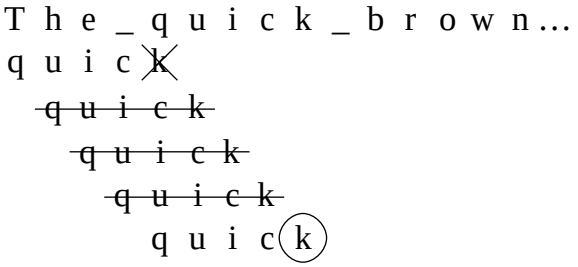
A related issue can arise because of inaccurate mappings from external character sets. To deal with this problem, characters that are easily confused by users can be kept in a weak equivalency class (đ *d-bar,* ð *eth,* Đ *capital d-bar,* Ð *capital eth*). This approach tends to do a better job of meeting 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 $O(m)$ 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 use only $O(m/n)$ in common cases, by skipping over characters in the target. Several implementers have adapted one of these algorithms to search text pretransformed according to a collation algorithm, which allows for fast searching with native-language matching (see *Figure 5-21*).

Figure 5-21. Sublinear Searching

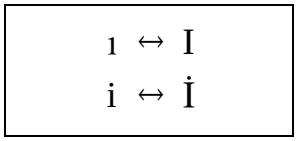


The main problems with adapting a language-aware collation algorithm for sublinear searching relate to multiple mappings and ignorables. Additionally, sublinear algorithms precompute tables of information. Mechanisms like the two-stage tables introduced in *Figure 5-1* are efficient tools in reducing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

5.18 Case Mappings

The vast majority of case mappings are uniform across languages. In a few instances, upper- and lowercase mappings may differ from language to language between writing systems that employ the same letters. 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 5-22*.

Figure 5-22. Case Mapping for Turkish I



The process of case mapping has important exceptions. See the file SpecialCasing.html on the CD-ROM for more about these exceptions. For more information on case mapping data, see *Section 4.1, Case—Normative*, and Unicode Technical Report #21, "Case Mappings," on the CD-ROM or the up-to-date version on the Unicode Web site.

Case Mappings Not Reversible. It is important to note that casing operations do not always provide a round-trip mapping. Also, because many characters are really caseless (most of the IPA block, for example), uppercasing a string does not mean that it will no longer contain any lowercase letters.

Case correspondences are not always one-to-one: the result of case folding may be a different character length than in the source string. For example, U+00DF ß LATIN SMALL LETTER SHARP S becomes "SS" in uppercase.

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).

Note that case transformations are not reversible. For example,

upper(lower("John Brown")) → "JOHN BROWN"

lower(upper("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*. Also, some single characters do not have reversible mappings. For example, U+03C2 ς GREEK SMALL LETTER FINAL SIGMA uppercases to U+03A3 Σ GREEK CAPITAL LETTER SIGMA, but the capital sigma lowercases to (nonfinal) U+03C3 σ GREEK SMALL LETTER SIGMA.

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 then return to it in 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

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 according to the actual appearance.

For information on case conversion, detecting when strings are of a given case, and performing caseless matching of strings, see Unicode Technical Report #21, "Case Mappings," on the CD-ROM or the up-to-date version on the Unicode Web site.

This PDF file is an excerpt from *The Unicode Standard, Version 3.0*, 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 (see http://www.unicode.org/unicode/uni2errata/UnicodeErrata.html). More recent versions of the Unicode standard exist (see http://www.unicode.org/unicode/standard/versions/).

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

ISBN 0-201-61633-5

This book is set in Minion, designed by Rob Slimbach at Adobe Systems, Inc. It was typeset using FrameMaker 5.5 running under Windows NT. ASMUS, Inc. created custom software for chart layout. The Han radical-stroke index was typeset by Apple Computer, Inc. The following companies and organizations supplied fonts:

Apple Computer, Inc.
Atelier Fluxus Virus
Beijing Zhong Yi (Zheng Code) Electronics Company
DecoType, Inc.
IBM Corporation
Monotype Typography, Inc.
Microsoft Corporation
Peking University Founder Group Corporation
Production First Software

Additional fonts were supplied by individuals as listed in the *Acknowledgments*.

The publisher offers discounts on this book when ordered in quantity for special sales. For more information please contact:

Corporate, Government, and Special Sales
Addison Wesley Longman, Inc.
One Jacob Way
Reading, Massachusetts 01867

Visit A-W on the Web: http://www.awl.com/cseng/

First printing, January 2000.