

# 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, dealing with this more complex behavior requires examining many aspects of programming practice and conventions in use today.

This chapter covers a series of short, self-contained topics that are useful to the implementer. The information and examples presented in this chapter will help implementers understand and apply the design and features of the Unicode Standard and will promote good practice in implementations conforming to the Unicode Standard. They are meant as guidelines and therefore are not binding on the implementer. The material is divided into three broad areas: basic programming, character semantics, and text handling.

---

### 5.1 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. Because of this, 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`) which can be compiled as `unsigned short` or `wchar_t` depending on the target compiler and platform. This allows 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` are implemented as 32-bit quantities, an implementation may transiently use 32-bit quantities to represent Unicode characters during processing. The internals 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*.

---

## 5.2 Compression and Transmission

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 be an attractive option. Compression always constitutes a higher-level protocol and makes interchange dependent on knowledge of the compression method employed.

**File-Based Compression.** There are commercially available compression algorithms, such as LZW, that, given enough context, will compress files to something near their theoretical minimum. Assume for example that a particular text takes 1000 bytes to encode in ASCII and that the compressed size is 437 bytes. When the same text is encoded using Unicode characters (taking 2000 bytes) and is then compressed properly, the resulting size will still be 437 bytes. Since Unicode-encoded text is composed from a 16-bit token set, algorithms such as LZW, which are sensitive to the width of the individual tokens, may stand to gain from being reimplemented based on 16-bit tokens.

Compression can be effective in eliminating the size overhead of a Unicode encoding without the cost of added complexity when it is built into the underlying support layer (such as modem transmission protocols or file system).

**String-Based Compression.** Occasionally it is necessary to compress short strings of text in isolation, or in a manner that allows access and substitution of text without decompression. In these specialized cases, string-based compression schemes must be used. However, compression employed at a level where it is visible to the text-processing parts of the program reintroduces the kind of complexities found in multibyte or other stateful encodings, which the Unicode character encoding was designed to avoid.

The Unicode codespace is arranged such that characters within the same script are generally contiguous, except for shared punctuation. A simple compression algorithm might run-length encode the most significant byte. Because of shared punctuation, better results are achieved by using two windows of 128 characters: one permanent window spanning the characters U+0000 → U+007F (equivalent to ASCII), and one sliding window whose start value can be set by escape codes.

Transformation forms defined in *Appendix A, Transformation Formats*, 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 using ASCII codes. On the other hand, text consisting of ideographs encoded with UTF-8 will require more space than equivalent Unicode-encoded text.

### 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 can be combined with compression in the same pass, thereby reducing the transmission overhead.

Alternatively, the UTF-7 and UTF-8 Transformation Formats described in *Appendix A, Transformation Formats*, may be used to transmit Unicode-encoded data through 7-bit or 8-bit transmission paths.

---

## 5.3 Language Information

The knowledge of the language of a given Unicode-encoded text is not explicitly specified by the character encoding itself; rather, a higher-level protocol is required to specify the language that a given text represents. Note that it may be possible to determine the language of a text based on its content heuristically, without any explicit language information. The accuracy of such a determination is based on the effectiveness of the heuristic. For example, text containing Hangul characters may reasonably be inferred to represent the Korean language since no other language employs the Hangul script. However, text containing Hebrew characters may or may not represent the Hebrew language since a number of unrelated languages use this script (for example, Hebrew (Ivrit), Yiddish, Judezmo (Ladino), and so on).

Language information is important to facilitate certain kinds of processing of character data. For example, correct searching, spell checking, and grammar checking of multilingual text certainly require language information to be marked for parts of the text. Even rendering may make use of language knowledge (see *Section 5.11, Strategies for Handling Non-Spacing Marks*).

The Unicode Standard does not encode language information. Therefore, if some aspect of character processing or display requires the discrimination of different uses of a single character according to language or writing system (that is, a particular linguistic and orthographic use of a particular character), a higher-level protocol is needed to specify the necessary language or orthographic bindings. An example of such a higher level protocol is the use of a language attribute as in SGML (Standard Generalized Markup Language) text encodings.

Languages in common use are covered by systems of language identifiers maintained by ISO as well as by vendors for use with their systems.

---



## 5.4 Unknown and Missing Characters

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

### *Unassigned and Private Use Character Codes*

There are two classes of character code values which 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. Some 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.<sup>1</sup>

### *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 , , , , , , , , , , , , , and so on.

---

## 5.5 Handling Surrogate Characters

Surrogates were carefully designed to be a simple and efficient extension mechanism that works well with older implementations and does not fall prey to many of the problems of multibyte encodings. They provide a mechanism for encoding extremely rare characters without requiring the use of 32-bit characters. Since only infrequently used characters will be assigned to surrogate pairs, implementations do not need to handle these pairs initially; vendors may choose to support them as market conditions require. *(No surrogate pairs are currently assigned, except for private use.)*

High surrogates and low surrogates are assigned to disjoint ranges of code positions. Non-surrogate characters can never be assigned to these ranges. Since 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 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 (not by a low surrogate or non-surrogate) and a high surrogate can be followed only by a low surrogate (not by a high surrogate or non-surrogate).

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 Unicode 1.0-conformant implementation as: "Hi <unrecognized><unrecognized>!!" This is only slightly worse than a Unicode 2.0 conformant implementation that did not support that surrogate pair, and so interpreted the sequence as: "Hi <unrecognized>!!"

So 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 is corrupted, the data corruption is localized (unlike some multibyte encodings like Shift-JIS or EUC); corrupting a single Unicode value affects only a single character. Because the high and low sur-

---

1. There are also some character code values that were assigned in a previous version of the Unicode Standard but that have become unassigned because the characters have been moved (see *Appendix D, Cumulative Changes*). Such code values should be recognized and converted into the correct Unicode 2.0 character code values where possible. In some cases, a Unicode 2.0 application may still need to emit Unicode 1.1 character codes to communicate with some Unicode 1.1 applications.



rogates are disjoint and always occur in pairs, this prevents the errors 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  $\alpha$  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 $\alpha$ corresponds to ■ ■ and to ■ ■.”
Weak	“The Greek letter $\alpha$ corresponds to ■ ■ and to <Phoenician>.”
Strong	“The Greek letter $\alpha$ 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 5.13, 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.

---

## 5.6 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 should be numerically interpreted as having 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 either be represented 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. This is another way of saying that you cannot do a 1-1 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 5-1 shows how the number 1,234 can be written.

**Figure 5-1. Ideographic Numbers**

一千二百三十四

or

一二三四

Supporting these digits for numerical parsing means that implementations have to be smart about distinguishing 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 5.14, 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.

---

## 5.7 Transcoding to Other Standards

The Unicode Standard exists in a world of other text and character encoding standards, some private, some national, some international. One of the major strengths of the Unicode Standard is the number of other important standards that it incorporates. In many cases, decisions on whether to unify characters or not were influenced by distinctions made in established and widely used standards, in order to allow round-trip transcoding.

Conversion of characters between standards is not always a straightforward proposition. There are many characters that have mixed semantics in one standard and may correspond

to more than one character in another. Sometimes standards give duplicate encodings for one and 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.

To assist and guide implementers, *The Unicode Standard, Version 2.0* provides a series of mapping tables on the accompanying CD-ROM. Each of these tables consist of one-to-one mappings from the Unicode Standard to another published character standard. They 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.

### **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 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.

### **Issues**

The Unicode Standard can be used as a pivot to transcode among  $n$  different standards. This reduces the number of mapping tables an implementation needs from  $O(n^2)$  to  $O(n)$ . Generally mapping *tables*—as opposed to algorithmic transformation—are required to map between the Unicode Standard and another standard. Table lookup yields much better performance than even simple algorithmic conversions, as can be implemented between JIS and Shift-JIS.

### **Tables and Virtual Memory**

Tables require space. Even small character sets often map to characters from several different blocks in the Unicode Standard, and so, in at least one direction, contain 64K entries.

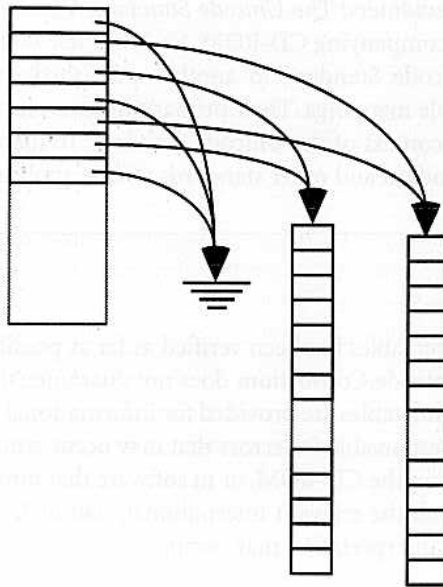
**Flat Tables.** If disk space is not at issue, virtual memory architectures have proven to yield acceptable working set sizes even for flat tables. This is because frequency of usage among characters differs widely and even small character sets contain many characters not used over large stretches of average text. Not only that, but 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 be transcoded to a given character set). This leaves large sections of the 64K sized reverse mapping tables (containing the default character, or unmappable character entry) unused—and therefore paged to disk. Similar results obtain for other types of tables needed to implement the Unicode Standard.

**Ranges.** It may be tempting to “optimize” these tables for space by providing elaborate provisions for nested ranges or similar devices. Given the branch penalties on modern, highly pipelined processor architectures, this leads to unnecessary performance penalties. A better way is a two- or three-stage table (trie) as described next, which can be coded without any test or branch instructions.

**Two-Stage Tables.** Two-stage (high-byte) tables are a commonly employed mechanism to reduce table size (see Table 5-2). 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 5-2. Two-Stage Tables**



**Optimized Two-Stage Table.** Wherever any blocks are identical, the pointers just point to the same block. For transcoding tables, this 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 provides access time that approaches the simple array access, but at a great savings in storage.

It is a simple matter, given an arbitrary 64K table, to write a small utility that can calculate the optimal number of stages and their width, but it is hard to improve on 8:8 two-stage tables because of their simplicity and ease of addressing 8 bits.

## 5.8 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 5.7, 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 as to 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 5.14, 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 per character in the input string.

In the past, many existing standards, like the C language standard, have assumed very minimalist "portable character sets" and geared their functions to operations on this set. As this practice is about to change and 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.

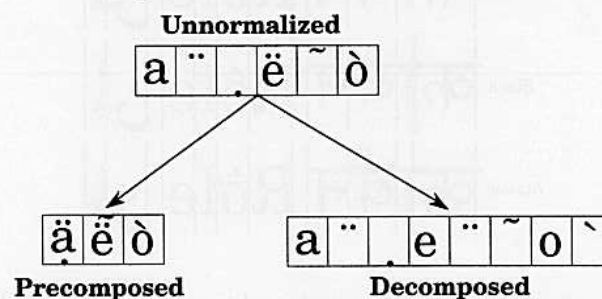
## 5.9 Normalization

**Converting to and from Canonical Form.** 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 non-spacing mark(s).

The Unicode Standard provides a table of normative spellings (maximal decompositions) of characters that can be composed of a base character plus one or more non-spacing marks. These tables can be used to unify spelling in a standard manner. Implementations that are 'liberal' in what they accept, but 'conservative' in what they issue, will have the least 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 also be added. The Unicode Standard, Version 2.0 includes the Tibetan script, which has a number of precomposed characters as well as combining marks; the decompositions for these new precomposed characters are included in the *Unicode Character Database* on the CD-ROM. Similarly, if a new combining mark is added to this standard, it may allow decompositions for precomposed characters that did not have decompositions before.

**Figure 5-3. Normalization**



**Normalization.** Systems may normalize Unicode-encoded text to one particular sequence, such as normalizing composed character sequences into precomposed characters or vice versa (see Figure 5-3).

Compared to the *possible* combinations, only a relatively small number of precomposed base character plus non-spacing marks have independent Unicode character values; most existed in dominant standards.

Systems that cannot handle non-spacing marks can normalize to precomposed characters, which provides for most modern Latin-based languages. Such systems can use fall-back rendering techniques to at least visually indicate combinations that they cannot handle (see the “Fall-back Rendering” subsection of *Section 5.12, Rendering Non-Spacing Marks*).

In systems that *can* handle non-spacing marks, it may be useful to normalize so as to eliminate precomposed characters. This allows such systems to have a homogeneous representation of composed characters and always have a consistent treatment of such characters. However, in most cases, it does not require too much extra work to support both forms, which is the simpler route.

Such systems may also normalize to a particular ordering of non-spacing marks. As stated in *Section 3.9, Canonical Ordering Behavior*, non-spacing marks that do not interact typographically can occur in any order after the base character. The Unicode Standard does not impose a particular order on non-interacting non-spacing marks for a number of reasons. The principle reason becomes clear when we consider a sequence that is “out of order.” In that case, the only reasonable interpretation is that it should have the same interpretation as the sequence in canonical order.

(For further information see *Chapter 3, Conformance*; *Chapter 4, Character Properties*; and *Section 2.5, Combining Characters*.)

---

## 5.10 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 leftwards-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 only be selected 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 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, there are two mechanisms used to visually indicate partial selection: linear and non-linear 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 afterwards.

This 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 non-spacing mark and drag to just to the left. It also does not allow the selection of individual non-spacing marks if there are more than one.

**Non-Linear 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 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) is 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, there is no single notion of editing character boundaries. At different times, users may want different degrees of granularity in the editing process. Two different methods suggest themselves. First, the user may set a global preference for the character boundaries. Second, the user may have alternate commands mechanisms, such as Shift-Delete, which are more (or less) fine than the default mode.

---

## 5.11 Strategies for Handling Non-Spacing Marks

By following these guidelines, a programmer should be able to implement systems and routines that provide for the effective and efficient use of non-spacing 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 or more sophisticated techniques that apply to more demanding situations, such as higher-end DTP (desktop publishing).

In this section, the terms *non-spacing mark* and *combining character* are be used interchangeably. The terms *diacritic*, *accent*, *stress mark*, *Hebrew point*, *Arabic vowel*, and others are sometimes used instead of the term *non-spacing mark*. (They refer to particular types of non-spacing marks.)

A relatively small number of implementation features are needed to support non-spacing 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.

Since non-spacing 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 non-spacing combining marks occurs, a glyph representing the combined form can be substituted. In simple, monospaced character rendering, a non-spacing 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 backwards. A trailing non-spacing 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 non-spacing marks or account for those cases where the composite character sequence has a different advance width than the base character, but such rendering systems are not necessary for the large majority of applications. Such systems can also supply more sophisticated truncation routines. (See also *Section 5.12, Rendering Non-Spacing Marks*.)

**Other Processes.** Correct multilingual comparison routines must already be able to compare a sequence of characters as one, 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 non-spacing marks in the target string which may affect the interpretation of the last matching character.

Line-break algorithms generally use state machines for determining word breaks. Such algorithms can also be easily adapted to prevent separation of non-spacing marks from base characters. (See also the discussion in *Section 5.15, Sorting and Searching*; *Section 5.9, Normalization*; and *Section 5.13, Locating Text Element Boundaries*.)

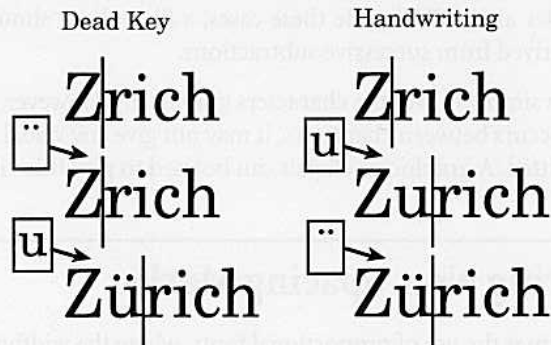
### **Keyboard Input**

A common implementation for the input of composed character sequences is the use of so-called *dead keys*. These match the mechanics used by typewriters to generate such sequences through overtyping the base character after the non-spacing 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. While 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 non-spacing marks (see Figure 5-5).

**Figure 5-5. Dead Keys Versus Handwriting Sequence**



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

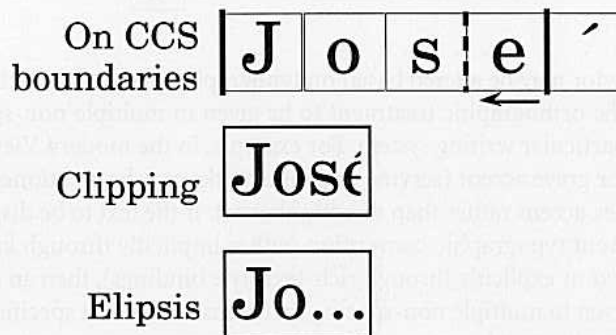
### Truncation

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

Truncation by character count is used where, due to storage restrictions, a limited number of characters can be entered into a field, or 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 composed character sequence boundary or even last word boundary) is often preferable to truncating after the last code element (see Figure 5-6). (See Section 5.13, *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 is on the basis of the width of the resulting string rather than a character count. In simple systems it is easiest to truncate by width, starting from the end and working back-

wards by subtracting character widths as one goes. Since a trailing non-spacing mark does not contribute to the measurement of the string, the result will not separate non-spacing 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, ligating, 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, this may look ugly, and 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.

---

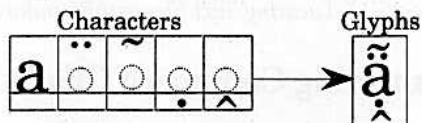
## 5.12 Rendering Non-Spacing Marks

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 only possible to get a semblance of a correct rendering in these fonts, especially with international characters.

In this section, the terms *non-spacing mark* and *combining character* are be used interchangeably. The terms *diacritic*, *accent*, *stress mark*, *Hebrew point*, *Arabic vowel*, and others are sometimes used instead of the term *non-spacing mark*. (They refer to particular types of non-spacing marks.)

When rendering a sequence consisting of more than one non-spacing mark, the non-spacing marks should, by default, be stacked outwards from the base character. That is, if two non-spacing marks appear over a base character, then the first non-spacing mark should appear on top of the base character, and the second non-spacing mark on top of the first. If two non-spacing marks appear under a base character, then the first non-spacing mark should appear beneath the base character, and the second non-spacing mark below the first (see Section 2.5, *Combining Characters*). This default treatment of multiple, potentially interacting non-spacing 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 based on knowledge of the specific orthographic treatment to be given to multiple non-spacing 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 non-spacing marks instead of that specified by the default inside-out rule just discussed.

**Fall-back Rendering.** There are several methods of dealing with an unknown composed character sequence which 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

Figure 5-8. Fallback Rendering



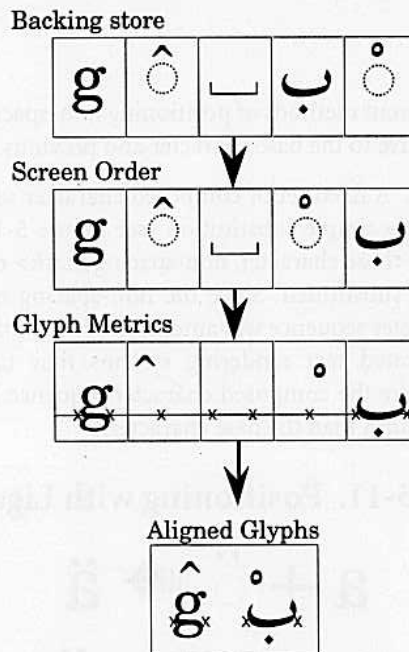
first and then rendering the non-spacing mark as an individual unit—with the non-spacing mark position on a dotted circle. (This convention is used in the Code Charts.)

Another method (*Simple Overlap*) uses default fixed positioning for an overlapping zero-width non-spacing 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 this will not be particularly attractive for letters such as *g-circumflex*, the result should generally be recognizable in the case of single non-spacing marks.

There is a degenerate case, where a non-spacing mark occurs as the first character in the text, or where it is separated from its base character by a *line separator*, *paragraph separator*, or other formatting character that causes a positional separation. In those cases, it is recommended that the non-spacing mark be shown with the *Show Hidden* alternative.

**Bidirectional Positioning.** In bidirectional text, the non-spacing 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.

Figure 5-9. Bidirectional Placement



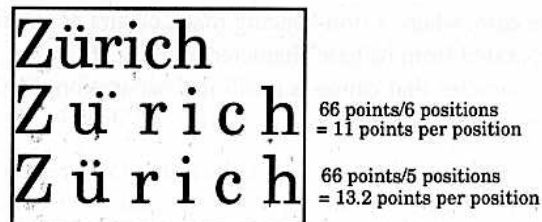
The simplest method is similar to the *Simple Overlap* fallback method. In the bidirectional algorithm, combining marks take the level of their base character. In that case, Arabic and Hebrew non-spacing marks would come to the left of their base characters. The font is designed so that instead of overlapping to the left, the Arabic and Hebrew non-spacing 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 non-spacing 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 have the non-spacing 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 non-spacing marks on the right of their base characters. Since the levels of non-spacing marks will be the same as their base characters, this pass can swap the order of non-spacing marks glyphs and base character glyphs in right-left (odd) levels. (See Section 3.11, *Bidirectional Behavior*.)

**Justification.** Typically, full justification of text adds extra space at space characters in order to widen a line; however, if there are too few (or no) space characters, some systems add extra letterspace between characters (see Figure 5-10). This process needs to be modified if zero-width non-spacing marks are present in the text. Otherwise, the non-spacing marks will be separate from their base characters.

Figure 5-10. Justification



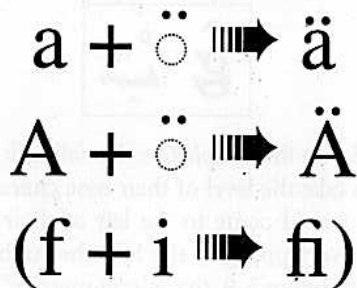
Since non-spacing marks always follow their base character, proper justification adds letterspace between characters only if the second character is a base character.

### Positioning Methods

There are a number of different methods of positioning non-spacing marks so that they are in the correct location relative to the base character and previous non-spacing 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, non-spacing mark> occurs, a glyph representing the combined form is substituted. Since the non-spacing 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



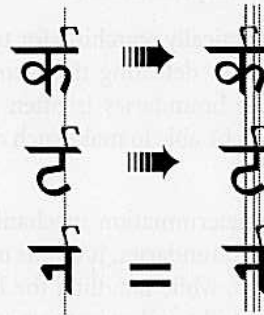


Positioning with ligatures is perhaps the simplest method of supporting non-spacing marks. Whenever there is a small, fixed set, such as the those corresponding to the precomposed characters of 8859-1 (Latin1), this method is straightforward to apply. Since the composed character sequence has the same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

If a composed character sequence does not ligate, 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 non-spacing marks is to use contextual formation (see Figure 5-12). In this case, there are several different glyphs corresponding to different positions of the accents. Base glyphs generally fall in to 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 non-spacing mark.

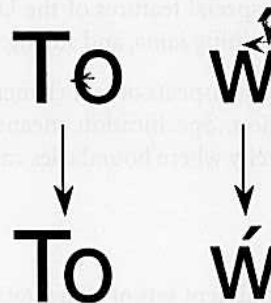
Figure 5-12. Positioning with Contextual Forms



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 get 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 is a process of mapping 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 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, since 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 get fine variations in position and shape.

---

## 5.13 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, 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, it simplifies computation to look 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 determination as quickly as possible.

The following example 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. Since 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.

However, for some languages this simple method is not sufficient. For example, Thai line-breaking 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 non-spacing 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 programmatically specify where boundaries can occur.

### **Boundary Specification**

A boundary specification defines different sets of characters, then lists the rules for boundaries in terms of those sets. The sets of characters 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

These elements take the following form in practice:

List	Meaning
a	Literal character a.
¬Y	Remove Y from the list
U+NNNN	Unicode character value, where N is an uppercase hexadecimal digit (0–9, A–F)
a-b	The range of all valid characters from the Unicode value a through b.
[YY]	Unicode character property from Unicode Character Database:
Lu	= Uppercase Letter
Li	= Lowercase Letter
Lt	= Titlecase Letter
Lm	= Modifier Letter (includes spacing versions of non-spacing marks)
Lo	= Other Letter
Mn	= Non-Spacing Mark
Mc	= Combining Mark (other than non-spacing; may reorder or surround)
Nd	= Decimal Number
No	= Other Number
Pd	= Dash Punctuation
Ps	= Open Punctuation
Pe	= Close Punctuation
Po	= Other Punctuation
Sm	= Math Symbol
Sc	= Currency Symbol
So	= Other Symbol
Zs	= Space Separator
Zl	= Line Separator
Zp	= Paragraph Separator
Cc	= Control or Format Character
Co	= Other Character (for example, private use)
Cn	= Non-Character (that is, not part of the Unicode Standard, Version 2.0)

A series of rules specifies a set of circumstances where a text element boundary can occur. The rules use fairly normal regular expression notation. The most interesting addition is a symbol that marks the position of a boundary, which can occur at any point in a rule.

Notation	Match
‡	position of boundary
¬X	characters not in X (or end-of-text or start-of-text)
X   Y	characters in either X or Y
X & Y	characters in both X and Y
X Y	a sequence of characters, the first in X, and second in Y
X*	zero or more characters in X
X+	one or more characters in X
{ X }	zero or one character in X
( X )	grouping, for application of above operations

There is always a boundary at the very start and at the very 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:

$$X Y^* ‡ \neg X \quad (2)$$

(In the text 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 position of any other boundaries between X and Y does 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 (1), this is more a limitation on the specification methods, since a rule with two boundaries could generally be expressed as two rules.

For example, a rule “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: in the following, the second set specification removes “AEIOUaeiou” from the first.

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

```
EngVowel = AEIOUaeiou
```

5. **No more than 256 sets.** This 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.

### 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. This would complicate each of the examples (and is so left as an exercise for the reader). In addition, it is intended that the rules themselves are 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 explained instead. An *underscore* (“\_”) is used to indicate a space in examples.

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.



## Character Boundaries

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 characters. Character representations should behave as a unit 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 is analogous to a system using conjoining *jamo* to represent Hangul syllables, where they are treated as single characters 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 character.

The principal requirements for general character boundaries are the handling of combining marks, Hangul conjoining *jamo*, and surrogate characters.

## Character Sets

¶	Paragraph Separator, Line Separator
Nsm	Non-spacing mark
L	Hangul leading jamo
V	Hangul vowel jamo
T	Hangul trailing jamo
Hs	High Surrogate
Ls	Low Surrogate

## Rules

Always break after paragraph separators, even when they are followed by combining marks. Also, always break before paragraph separators (but that is caught by rule 2 anyway).

$$¶ \neq \quad (1)$$

Break before base characters (non-combining characters). Notice that it is also necessary to exclude some other characters that might not break from preceding characters, depending on other context.

$$\neq \neg(Nsm | Ls | L | V | T) \quad (2)$$

Break around Hangul syllables. The first rule breaks before Hangul syllables, while the others break except within an allowable sequence. Notice that there are no breaks before combining marks, so that they can apply to Hangul syllables.

$$\neg(L | V | T) \neq L | V | T \quad (3)$$

$$L \neq \neg(L | V | T | Nsm) \quad (4)$$

$$V \neq \neg(V | T | Nsm) \quad (5)$$

$$T \neq \neg(T | Nsm) \quad (6)$$

Break around isolated surrogates (degenerate cases). Note to keep combining marks from applying to an isolated surrogate.

$$\neg Hs \neq Ls \quad (7)$$

$$Hs \neq \neg Ls \quad (8)$$

$$\neg Hs Ls \neq \quad (9)$$

**Note:** The only combining marks that are not non-spacing marks are certain Indic matras. These characters do not have the simple behavior of non-spacing marks; Indic clusters (such as defined in the Devanagari subsection in Section 6.1, *General Scripts Area*) have a

more complex structure and may or may not be considered by users to be single characters. Where they are, the preceding rules would be amended to not break within sequences of the following form (where Con = consonant & Cm = combining mark):

$$\text{Con (Virama Con)}^* \text{Cm}^*$$

**Word Boundaries.** Word boundaries are used in a number of different contexts. The most familiar ones are for 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 non-letters. Trailing spaces cannot be counted as part of a word, since then searching for “abc” would fail on “abc\_”.

For word selection, the rules are somewhat less clear. Some programs include trailing spaces, while others include neighboring punctuation. Where words do not include trailing spaces, sometimes programs treat the individual spaces as separate words; other times they treat a whole 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 to “The\_fox”.

This discussion outlines the case where boundaries occur between letters and non-letters, and there are no boundaries between non-letters. The discussion 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.

## Character Sets

¶	Paragraph Separator, Line Separator
Let	Letter
Mid	hyphen, apostrophe (“)
Nsm	non-spacing mark
Hira	Hiragana
Kata	Katakana
Han	Han ideograph (Kanji)

## Rules

Always break after paragraph separators.

$$\text{¶} \quad \# \quad (1)$$

Break between letters and non-letters. Include trailing non-spacing marks as part of a letter.

$$\neg(\text{Let} \mid \text{Nsm}) \quad \# \quad \text{Let} \quad (2)$$

$$\text{Let Nsm}^* \quad \# \quad \neg\text{Let} \quad (3)$$

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 non-spacing marks.

$$\neg(\text{Hira} \mid \text{Kata} \mid \text{Han} \mid \text{Nsm}) \quad \# \quad \text{Hira} \mid \text{Kata} \mid \text{Han} \quad (4)$$

$$\text{Hira Nsm}^* \quad \# \quad \neg\text{Hira} \quad (5)$$

$$\text{Kata Nsm}^* \not\# \neg(\text{Hira} | \text{Kata}) \quad (6)$$

$$\text{Han Nsm}^* \not\# \neg(\text{Hira} | \text{Han}) \quad (7)$$

- One could also generally break between any letters of different scripts. In practice—except for languages that do not use spaces—this 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 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 non-spacing marks should not be separated from the base character.

Non-spacing marks may be exhibited in isolation; that is, over a space or non-breaking space. In that case, the the whole composed character sequence is treated as a unit. If the composed character sequence consists of a *no-break space* followed by non-spacing marks, then it does not generally allow linebreaks before or after the sequence. If the composed character sequence consists of any other space followed by non-spacing marks, then it generally does allow linebreaks before or after the sequence.

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

### Character Sets

¶	Paragraph Separator, Line Separator
Sp	Space separator
Nb	Non-breaking space, non-breaking hyphen, zero-width non-breaking space
Nsm	Non-spacing mark
Ideo	Hiragana, Katakana, Bopomofo, Han
Open	Open Punctuation
Close	Close Punctuation, Period, Comma,...

### Rules

Always break after paragraph separators.

$$\text{¶} \not\# \quad (1)$$

Break between trailing spaces and other characters. Include non-spacing marks, since a non-spacing mark applied to a space is used to allow presentation of the non-spacing mark in isolation.

$$\text{Sp Nsm}^* \not\# \neg(\text{Sp} | \text{Nsm} | \text{Nb}) \quad (2)$$

Handle ideographs specially. Break around ideographs except for opening and closing punctuation characters (also known as *taboo* characters) and non-spacing marks.

$$\text{Ideo Nsm}^* \not\# \neg\text{Close} \quad (3)$$

$$\neg(\text{Open Nsm}^*) \not\# \text{Ideo} \quad (4)$$

**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).

## Character Sets

¶	Paragraph Separator, Line Separator
Sp	Space separator
Term	!?
Dot	Period
Cap	Uppercase, Titlecase & non-cased letters
Lower	Lowercase

## Rules

Always break after paragraph separators.

$$\text{¶} \quad \# \quad (1)$$

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

$$\text{Term Close}^* \text{Sp}^* \{\text{¶}\} \quad \# \quad (2)$$

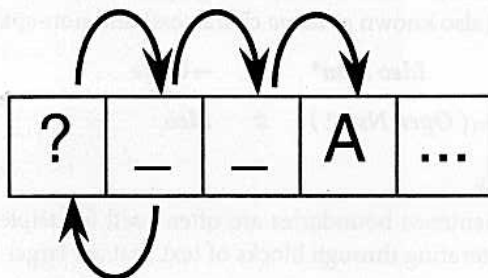
Handle period specially, since it may be an abbreviation or numeric period—and not the end of a sentence. Don't break if it is followed by a lowercase letter instead of uppercase.

$$\text{Dot Close}^* \text{Sp}^+ \quad \# \quad \text{Open}^* \neg \text{Lower} \quad (3)$$

**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 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 backwards with this second set of rules until a safe starting point is located, then iterate forwards from there. Iterate forwards to find boundaries that were between the starting point and the safe point; discard these. The desired boundary is the first one which 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 as to 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.

---

## 5.14 Identifiers

A common task facing an implementer of the Unicode Standard is the provision of a parsing and/or lexing engine for identifiers. In order to assist in the standard treatment of identifiers in Unicode character-based parsers, the following guidelines are provided for the definition of identifier syntax.

Note that the task of parsing for identifiers and the one of locating word boundaries are related tasks and it is straightforward 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 starts with a letter or an ideograph, and then follows with 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 only need to 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
3. Inclusion of extenders such as U+00B7 MIDDLE DOT
4. Inclusion of U+FF3F FULLWIDTH LOW LINE as equivalent to U+005F LOW LINE

**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 results from the conformance rules in *Chapter 3, Conformance*, regarding interpretation of canonical-equivalent character sequences.

The four enclosing combining marks (U+20DD → U+20E0) are given special treatment in the syntactic definition of <ident\_combining\_char> 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 alternate 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 assigned to an ignorable character class for the purposes of identifier definition. Some implementations may choose to filter out these ignorable characters; this has the advantage that two identifiers that appear to be identical will more likely *be* identical.

**Extenders.** U+00B7 MIDDLE DOT participates in the canonical decomposition of Unicode characters that are otherwise valid as elements of identifiers (for example, U+013F LATIN CAPITAL LETTER L WITH MIDDLE DOT). Treating middle dot as a letter extender that is valid in an identifier is consistent with the treatment of modifier letters as valid elements of identifiers and also parallels the treatment of letter plus diacritic combining marks.

**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 underscore as an identifier part (but not an identifier start); C identifiers allow underscore for either an identifier part or an identifier start. The syntax defined next shows inclusion of underscore for identifier part to illustrate that the <underscore> class should include U+FF3F FULLWIDTH LOW LINE, as well as the ASCII-derived underscore (U+005F LOW LINE) common to many character sets.

### Terminology

- <x> indicates a named syntactic entity
- {x} indicates a set, defined either by specification of character properties, or by enumeration of coded character values
- x-y indicates set subtraction
- +/[x] indicates presence or absence of the character property within the brackets
- (x) indicates grouping
- x\* indicates occurrence of a sequence of zero or more instances of a syntactic entity

### Terminal Classes

<alphanumeric_char>	::= {+[alphanumeric]}
<initial_alphanumeric_char>	::= {+[alphanumeric] -[combining]}
<combining_char>	::= {+[combining]}
<ideographic_char>	::= {+[ideographic]}
<decimal_digit_char>	::= {+[decimaldigit]}
<enclosing_char>	::= { 20DD, 20DE, 20DF, 20E0 }
<zw_layout_char>	::= { 200C, 200D, 200E, 200F }
<bidirectional_format_char>	::= { 202A, 202B, 202C, 202D, 202E }
<alt_format_char>	::= { 206A, 206B, 206C, 206D, 206E, 206F }
<zw_nonbreak_space>	::= { FEFF }
<underscore>	::= { 005F, FF3F }
<extender>	::= { 00B7, 02D0, 02D1, 0387, 0640, 0E46, 0EC6, 3005, 3031..3035, 309B..309D, 309E, 30FC..30FE, FF70, FF9E, FF9F }

### Syntactic Rules

<ident_combining_char>	::= <combining_char> - <enclosing_char>
<ident_ignorable_char>	::= <zw_layout_char>   <bidirectional_format_char>   <alt_format_char>   <zw_nonbreak_space>
<identifier_start>	::= <initial_alphanumeric_char>   <ideographic_char>

```

<identifier_part> ::= <alphabetic_char> | <ideographic_char> |
                   <decimal_digit_char> | <ident_combining_char> |
                   <underscore> | <extender> | <ident_ignorable_char>
<identifier> ::= <identifier_start> ( <identifier_part> )*

```

### Character Properties

The exact list of characters with a given property is given in *Chapter 4, Character Properties*.

[decimaldigit] is a normative property of the Unicode Standard.

[combining] is a normative property of the Unicode Standard.

[ideographic] is an informative property of: the Unified Han set (U+4E00 → U+9FA5); the Compatibility Han characters (U+F900 → U+FA2D); U+3007 IDEOGRAPHIC NUMBER ZERO; and the Hangzhou-style numerals, (U+3021 → U+3029).

Other Unicode characters may involve ideographs, but these are either treated as neutral symbols (for example, circled ideograph symbols) or require special formatting that disqualifies them from participation in identifiers (for example, Kanbun symbols).

[alphabetic] is a informative property of the primary units of alphabets and/or syllabaries, whether combining or non-combining; composite characters that are canonically equivalent to a combining character sequence of an alphabetic base character plus one or more combining characters; letter digraphs; contextual variants of alphabetic characters; ligatures of alphabetic characters; contextual variants of ligatures; modifier letters; letterlike symbols that are compatibility equivalents of single alphabetic letters; and miscellaneous letter elements, notably U+00AA FEMININE ORDINAL INDICATOR and U+00BA MASCULINE ORDINAL INDICATOR.

---

## 5.15 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 their proximity 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.

### 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 look-up 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. Swedish

treats U+00C4 LATIN CAPITAL LETTER A WITH DIAERESIS as an individual letter, sorting it after *z* in the alphabet; however, German 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 impact on culturally expected sorting.)

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

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 of this scheme looks very similar to the decomposition of Unicode characters into base character and accent. The fact that the Unicode Standard allows 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
- grouped or expanding characters
- ordering of unmapped characters
- more than one level of ignorable characters.

### ***Unicode Character Equivalence***

Section 3.6, *Decomposition*, and Section 3.9, *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 as exactly the same, no matter what the underlying storage is. Figure 5-15 gives two examples of this.

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

### ***Similar Characters***

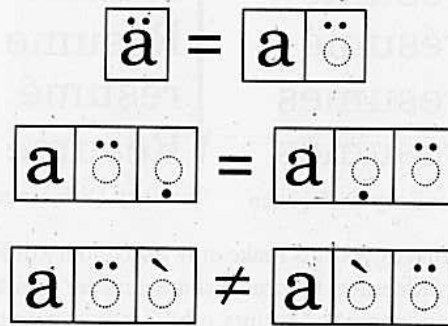
Languages differ in what they consider similar characters, but users generally want characters that are similar (such as upper- and lowercase) to sort close to each other but not to be

---

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.



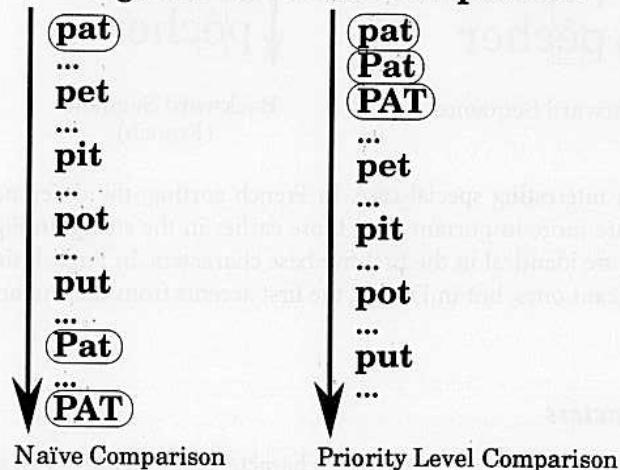
Figure 5-15. Character Equivalence



collated exactly the same way. Were upper- and lowercase to collate identically, words differing only in case would appear in random order (see Figure 5-16). The same is true for treating 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 only applies when the strings are the same in all other respects. If this were not so, *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

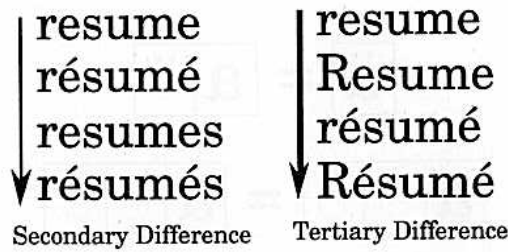


### Levels of Comparison

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 these rules:

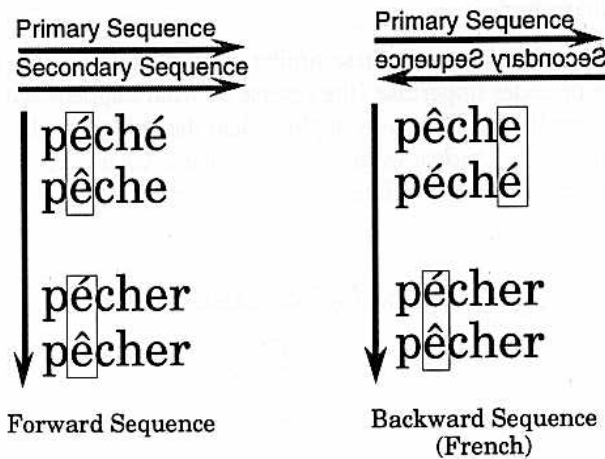
- R1 Count secondary differences—only if there are no primary differences.
- R2 Count tertiary differences—only if there are no primary or secondary differences.

**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.

**Figure 5-18. Orientation**

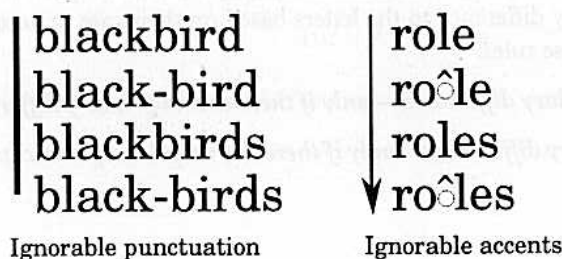


French presents an interesting special case. In French sorting, the differences in accents later in the string are more important than those earlier in the string. In Figure 5-18, the two pairs of words are identical in the first five base characters. In English the first accents are the most significant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

**Ignorable Characters**

Another class of interesting cases are 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**



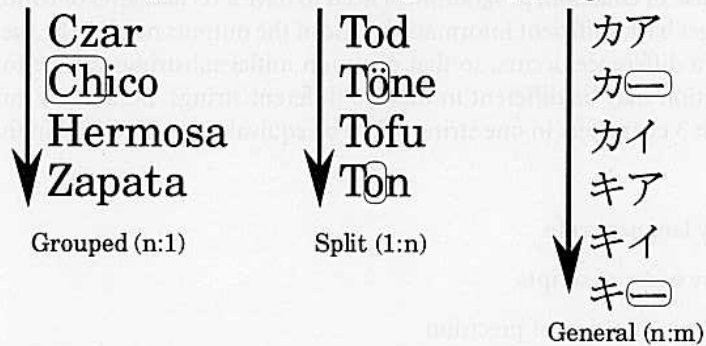
The general rule is

**R3** *Treat ignorable characters as a primary or secondary 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 can compare as if they were one character (see Figure 5-20). In traditional Spanish orthography, “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 — *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 — *length mark* indicates a long “a” and comes after ア “a”, while after the character キ “ki” the — *length mark* indicates a long “i”, and comes after イ “i”.

Look at the second character in each word in the third column of the figure. There are three different characters in the second position: ア “a”, — *length mark*, and イ “i”. The general rule is

**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 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 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 of the outputs 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 3 characters in one string might be equivalent to the first 4 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 strings for fast comparison
- Or process just as much as needed for stand-alone

### ***Optimizations***

Multiple-level comparison requires a bit more work than binary comparison. While real life studies put the overhead at around 50 percent, it often pays to first transform terms to be sorted into equivalent *sortkeys*, 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 sortkeys 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, sortkeys 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.

Sortkeys 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 reduces computation 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. Additional features are often added, such as only matching words (that is, there is a word boundary on each side of the match). One technique is to code a



fast search for a weak match. When a candidate is found, then additional tests can be made for additional criteria (such as matching diacriticals, word match, case match, and so on).

When searching strings, remember to check for trailing non-spacing marks in the target string that may affect the interpretation of the last matching character. That is, if you search for “San Jose”, you may find a match in the string “Visiting San José, Costa Rica is a...” If you are searching for an exact (diacritic) match, then you want reject this match. If you are searching for a weak match, then you want to accept the match, but be sure to include any trailing non-spacing marks when you return the location and length of the target substring. The mechanisms discussed in *Section 5.13, Locating Text Element Boundaries*, can be used for this.

Once important 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*, and the Latin subsection in *Section 6.1, General Scripts Area*). The preferred method of implementing case insensitivity uses the same mechanisms and tables as discussed 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, *dotless-i*, *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, characters that are easily confused by users can be kept in a weak equivalency class (*d-bar*, *eth*, *capital d-bar*, *capital eth*). This 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 represents 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 searched for, and  $m$  is the number of characters in the target to be searched.

There are a number of algorithms that 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 pre-transformed according to a collation algorithm, which allows for fast searching with native-language matching (see Figure 5-21).

**Figure 5-21. Sublinear Searching**

```

T h e _ q u i c k _ b r o w n ...
q u i c k
q u i e k
q u i e k
q u i e k
q u i c (k)

```

The main problems with adapting language aware collation algorithm for sublinear searching are caused by multiple mappings and ignorables. Additionally, sublinear algorithms precompute tables of information. Mechanisms like the two-stage tables introduced in Figure 5-2 are efficient tools in reducing memory requirements.