

## 2.0 *General Principles of the Unicode Standard*

This chapter discusses the fundamental principles governing design of the Unicode standard. It includes discussion of text processes, unification principles, allocation of codespace, character properties, and a detailed definition and description of non-spacing marks and how they are employed in Unicode character encoding. This chapter also details the formal requirements for creating a text processing system that conforms to the Unicode standard.

### 2.1 Architectural Context of a Character Encoding

Character codes are like nuts and bolts—they are minor components that hold together a host of different useful systems.

A character code standard such as the Unicode standard enables the implementation of useful processes operating on textual data. The interesting end products are not the character codes but the text processes, since these directly serve the needs of system users.

#### *Text Elements, Code Elements, and Text Processes*

One of the more profound challenges in designing a worldwide character encoding stems from the fact that languages differ in what they consider a fundamental unit of text, or *text element*.

For example, in Spanish, the combination “ll” is a text element for the process of sorting but not for the process of rendering; in Croatian, the objects “lj” and “l-followed-by-j” are distinct text elements for transliteration, but not for the process of rendering; and in English, the objects “A” and “a” are distinct text elements for the process of rendering but generally not for the process of spell-checking. Notice that the text elements in a given language depend upon the specific text process.

A character encoding standard has fundamental units of encoding, *code elements* or characters, which must exist in a unique relationship to the assigned *code points*.

The design of the character encoding scheme must provide precisely that set of code elements which allows programmers to design applications that can perform text processes in the languages they intend to support.

Most computer systems provide low-level functionality for a small number of basic text processes, out of which more sophisticated text-processing capabilities are built. The following is a list of text processes used by most computer systems:

- Rendering characters visible (including ligatures, contextual forms and so on)
- Breaking lines while rendering (including hyphenation)
- Computing directionality
- Modifying appearance, such as kerning, underlining, slant, and boldface
- Determining units such as “word,” and “sentence”
- Interacting with users in processes such as resolving mouse selection and highlighting text
- Modifying keyboard input, and editing stored text through insertion and deletion
- Comparing text in operations such as determining sort order of two strings, or filtering or matching strings
- Analyzing text content in operations such as spell-checking, hyphenation, and parsing morphology
- Treating text as bulk data for operations such as compressing and decompressing, truncating, and transmitting and receiving

In the case of an English encoding such as ASCII, the relationships between the encoding and the basic text processes built on it are seemingly straightforward: Characters are rendered visible one by one in defined rectangles from left to right in a linear order. Thus one character code inside the computer corresponds to one logical character in a process such as simple English rendering.

When designing an international and multilingual text encoding system such as the Unicode standard, the relationship between the encoding scheme and implementation of basic text processes must be considered explicitly, for several reasons:

- Some or all of the assumptions about character rendering that hold true for English fail for other writing systems. Outside of English, characters are not necessarily rendered visible one by one in rectangles from left to right. In many cases, character positioning is quite complex and does not proceed in a linear fashion.
- The set of text characters appropriate for encoding a language is often debatable. For example, there is disagreement about the encoding of commonly used characters in French and German: ISO 8859 defines letters such as “à” and “ü” as individual characters, whereas ISO 6937 represents them by composition instead. While this reflects the difference in the identity of these characters between German and French, neither encoding is ideally suited to handle both.

- No encoding can include all basic text processes equally well. As a result, some trade-off is necessary. For example, ASCII defines separate codes for upper- and lowercase letters. This causes some text processes, such as rendering, to be carried out more easily, and some processes, such as comparison, to be more difficult. A different encoding design for English, such as case-shift control codes would have made the opposite true. In designing a new encoding for complex scripts, such tradeoffs must be evaluated, and decisions made explicitly, rather than unwittingly.

The design of the Unicode encoding scheme is independent of the design of basic text processing algorithms, with the exception of directionality (see Appendix A). Unicode implementations are assumed to contain suitable text processing and/or rendering algorithms, which may be more or less complex. In particular, sorting and string comparison algorithms cannot assume that the assignment of Unicode character code numbers provides an alphabetical ordering for lexicographic string comparison. In general, the culturally expected sorting orders require arbitrarily complex sorting algorithms. The expected sort sequence for the same characters differs across languages, so in general no single linear ordering exists. The Unicode standard does not assume any particular string comparison process, but its design does assume the capability to implement sufficiently powerful algorithms.

There is no reason to expect text processes in general to be as simple as they are for English. Nevertheless, a computer system that can offer its users highly sophisticated operating and graphical windowing environments can also be sophisticated enough to transmit and render text in a user's own script and language.

### *Plain and Fancy Text*

*Plain text* is a pure sequence of character codes; plain Unicode text is a sequence of Unicode character codes. *Fancy text* is any text representation consisting of plain text plus added information such as font size, color, and so on. For example, a multifont text as formatted by a desktop publishing system is fancy text.

The kinds of data structures that can be built into fancy text are of many possible types. To give but one example, in fancy text containing ideographs, it would be possible to store the phonetic reading of each ideograph somewhere in the text structure. On the other hand, the simplicity of plain text gives it a natural role as a major structural element.

Both plain and fancy text are already familiar constructs in ASCII-based systems and their relative functional roles are well known:

- Plain text is public, standardized, and universally readable.
- Fancy text is often intended to be private, implementation-specific, and is often proprietary.
- Plain text is the underlying *content* stream to which formatting can be *applied*.

The details of any particular fancy text design can be made public or standardized, but the fact remains that most fancy text designs are vehicles for particular implementations, and are not readable by other implementations. Since fancy text equals plain text plus added information, the extra information in fancy text can always be stripped away to reveal the “pure” text underneath. This operation is familiar, for example, in word processing systems that deal with both their own private fancy format and with the universal plain ASCII text file format. Thus by default, plain text represents *the basic, interchangeable content of text*.

Since plain text represents character content, plain text as such has no appearance at all. It requires a rendering process to make it visible.

If the same plain text sequence is given to disparate rendering processes, there is no expectation that rendered text in each instance should have the same appearance; all that is required from disparate rendering processes is to make the text legible according to the intended reading. Therefore, the relationship between appearance and content of plain text may be stated as follows: *Plain text must contain enough information to permit the text to be rendered legibly, nothing more.*

## 2.2 The Basic Design of the Unicode Character Encoding

This section presents the basic principles which have served to guide the overall design of the Unicode standard. It also clarifies the distinction between encoding characters and glyphs.

### *Unicode Principles*

Design of the Unicode standard reflects the following principles. Not all principles can be satisfied simultaneously. While every effort has been made to maintain consistency for the sake of simplicity and efficiency, there were many cases where exceptions were made to maintain compatibility with existing standards.

1. All Unicode characters have a uniform width of 16 bits. Plain Unicode text consists of pure 16-bit Unicode character streams (in files or strings).
2. The full codespace (over 65,000 characters) is available to represent characters. ISO 646 and 8859 control code positions have been retained for compatibility, and a number of codes have been reserved for use as signals in the text stream.
3. Characters are made visible through a rendering process which (at its lowest level) requires that characters be mapped to glyphs. The default rendering order of Unicode text characters is logical (keystroke) order.
4. The Unicode standard allows dynamic composition of accented forms or static composed forms. Common static-form single character codes such as LATIN CAPITAL U WITH DIAERESIS “Û” are included for compatibility with current international standards. However, because the



process of character composition is open-ended, additional letters with modifying non-spacing marks can be created from a combination of base letters and non-spacing marks.

Unicode non-spacing marks (accents and so on) used to create composite forms are *productive*. This means that they are *in practice* combined with a large number of base form characters. For example, the diaeresis, “¨”, can be combined with all vowels and a number of consonants in languages using the Latin script or others. The stroke used in the Polish LATIN CAPITAL LETTER L SLASH, however, is of limited use, hence LATIN CAPITAL LETTER L SLASH is always treated as a single unit and not composed of two distinct characters. This productivity extends to non-Roman languages as well.

5. The Unicode standard encodes characters in scripts which can be used for a number of different natural languages. Punctuation and symbols which are common across scripts are given a single code.
6. The Unicode standard avoids duplication of characters by unifying them across languages: characters that are equivalent in form, usage, and essential properties are given a single code. Common letters, punctuation marks, symbols, and diacritics are given one code each, regardless of language, as are common Chinese/Japanese/Korean (CJK) ideographs. Letters, symbols, and CJK ideographs with common shapes but different functions are given separate codes. Other than what is required to preserve plain text distinctions, the Unicode standard does not attempt to encode features such as language, font, size, positioning, glyphs, and so forth. For example, it does not preserve language as a part of character encoding: Chinese “zi” (字), Japanese “ji” (字) and Korean “ja” (字) are all represented as the same character code, as are French “i grecque” (Y), German “ypsilon” (Y), and English “wye” (Y).
7. The distinction between the Unicode standard and other forms of data (ASCII, pictures, and so on) is the function of a higher-level protocol (text classes and layout) and not specified by the Unicode standard itself. The 64 control code positions of ISO 646 and 8859 are retained only for compatibility.
8. Character identity is preserved over a number of different combined national standards. Where variant forms are given separate codes within one included standard, they are also kept separate within the Unicode standard. This guarantees that there will always be a mapping between the Unicode standard and included national standards.
9. In determining whether or not to unify ideographic variant forms across standards, the Unicode standard follows the guidelines published by JIS. These guidelines are found in the Japan Industrial Standard *Jouhou koukan you kanji fugoukei* (Code of the Japanese Graphic Character Set for Information Interchange). C 6226-1983 §3.4 *Kanji no itaiji no toriatsukai* (The handling of variant Han characters.) Though written with Japanese usage in mind, they are general enough to be applied to Chinese and Korean as well. Where these guidelines suggest that two

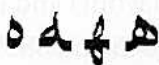
forms constitute a trivial (*wazukana*) difference, the Unicode standard assigns a single code. Otherwise, separate codes are assigned.

Conversion between Unicode text and text in other character codes must be done by explicit table-matching processes. There is no guaranteed bit-for-bit compatibility with other codes, even though accurate convertability is guaranteed between Unicode and other widely accepted international standards as of December 1, 1990. Unicode text may be compressed for storage or transmission like any other binary data, but the Unicode standard specifies no preferred compression algorithm and guarantees no bit-for-bit identity in compressed format.

### *Glyphs*

The Unicode standard was designed to encode characters. There are various relationships between character and glyph: a single glyph may correspond to a single character, or to a number of characters, or multiple glyphs may result from a single character.

“Glyphs” may be considered as discrete components of writing. The distinction between characters and glyphs is illustrated in the following table:

<i>Glyph</i>	<i>Unicode Character(s)</i>
A	U+0041 LATIN CAPITAL LETTER A
fi, fi	U+0066 + U+0069 LATIN SMALL LETTER F followed by LATIN SMALL LETTER I
à, à, à	U+0061 LATIN SMALL LETTER A
	U+0647 ARABIC LETTER HEH (positional forms)

Unicode characters represent primarily, but not exclusively, the letters, punctuation, and other signs that comprise natural language text and scientific and technical documentation. Characters reside only in the machine, as strings in memory or on disk, in the backing store. The Unicode standard deals only with character codes. In contrast to characters, glyphs appear on the screen or paper as particular representations of one or more backing store characters. A repertoire of glyphs comprises a font.

Glyph shape and glyph identifier assignments are the responsibility of individual font vendors and of the glyph identifier standards.

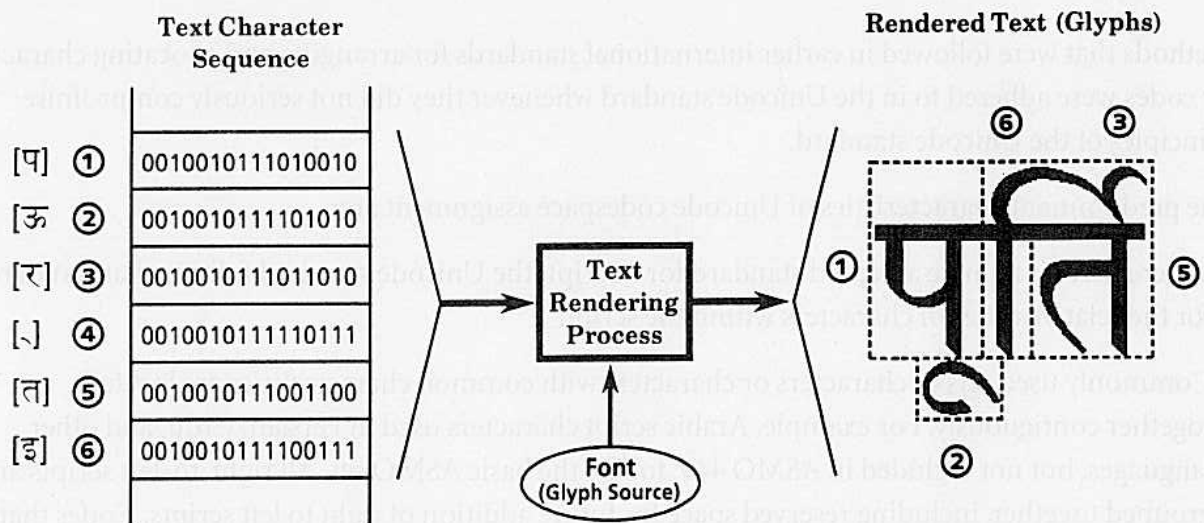


Figure 2-1. Unicode Character Code to Rendered Text Glyph

The process of mapping from characters in the backing store to glyphs is one aspect of text rendering. The final appearance of rendered text is dependent on context (neighboring characters in the backing store), variations in typographic design of the fonts used, and formatting information (point size, superscript, subscript, and so on). The results on screen or paper can differ considerably from the expected or prototypical shape of a letter or character. The glyph “A” displayed on the screen must not be confused with the character “A” in the backing store.

### 2.3 Codespace Allocation

All codes in the Unicode standard are equally accessible electronically; the exact arrangement of codes is of minor consequence for information processing. But, for the convenience of people who will use them, the codes are grouped by linguistic and functional categories.

Codespace in the Unicode standard is divided into six zones: General Scripts (alphabetic and other scripts that have relatively small character sets), Symbols, CJK (Chinese, Japanese, and Korean) Auxiliary, CJK Ideographs, Private Use, and Compatibility.

The General Script zone covers alphabetic or syllabic scripts such as Latin, Cyrillic, Greek, Hebrew, Arabic, Devanagari and Thai. The Symbol zone includes a large variety of characters for punctuation, mathematics, chemistry, dingbats, and so on. The CJK Auxiliary zone includes punctuation, symbols, Kana, Bopomofo, and single and composite Hangul. The CJK Ideographic zone currently provides space for over 20,000 ideographic characters or characters from other scripts. The Private Use zone (5,632 code points) is used for defining user- or vendor-specific graphic characters. The Compatibility Zone contains characters from widely used corporate and national standards that have other canonical representations in Unicode encoding. (See figure 1-1 for an overview of Unicode codespace allocation.)

Methods that were followed in earlier international standards for arranging and allocating character codes were adhered to in the Unicode standard whenever they did not seriously compromise principles of the Unicode standard.

The predominant characteristics of Unicode codespace assignment are:

- Where there is a single accepted standard for a script, the Unicode standard follows that standard for the relative order of characters within the script.
- Commonly used sets of characters or characters with common characteristics are located together contiguously. For example, Arabic script characters used in Persian, Urdu, and other languages, but not included in ASMO 449, follow the basic ASMO set. All right-to-left scripts are grouped together, including reserved space for future addition of right to left scripts. Codes that represent letters, punctuation, symbols, and diacritics that are shared by multiple languages are grouped together.
- The Unicode standard makes no pretense to correlate character encoding with collation or case. Even in ASCII, raw character codes alone are not sufficient for collating. Upper- and lowercase correlation is language dependent.
- The first 256 codes follow precisely the arrangement of ISO 646 (ASCII) and ISO 8859-1 (Latin 1).
- Chinese, Japanese and Korean phonetic symbols are grouped together by language in standard order.
- Only sixty-five codes (U+0000→U+001F and U+007F→U+009F) are reserved specifically as control codes. U+FFFF and U+FFFE are reserved and should not be transmitted or stored. All other code points are reserved for graphic characters. Null (U+0000) can be used as a string terminator as in the C language.
- 5.5K of user space has been allocated in the range U+E800→U+FDFF. There is no escape mechanism for extension into a larger codespace, so it is not necessary to test every character for escape sequences.
- Code points unassigned in the Unicode standard, version 1.0 are available for assignment in later versions of the Unicode standard to characters of any script. Existing characters will not be re-assigned or removed.

## 2.4 Character Properties, Controls, and Sequences

This section provides an overview of character properties, control characters, and bidirectional character ordering.



## Properties

Character properties tables are provided for use in parsing, sorting, and other algorithms requiring semantic knowledge about the code points. The properties identified by the Unicode standard include: digits, numbers, space characters, non-spacing marks, and direction. Tables of character properties are located in Chapter 4.

## Control Characters

The Unicode standard provides ranges of codespace for the representation of control characters, which are not to be used for graphic characters. These ranges are U+0000→U+001F and U+007F→U+009F, which correspond to the 8-bit controls 00 to 1F (C0 controls) and 7F to 9F (*delete* and C1 controls); the Unicode values are simply zero-extended from the 8-bit values. For example, the 8-bit version of HT (horizontal tab) is at 09: the Unicode standard 16-bit version of HT is at U+0009.

Some systems may use a sequence of characters beginning with a control code to encode additional information about text, such as formatting attributes or structure. These sequences can be represented in a Unicode encoding, but must be represented in terms of 16-bit characters. For example, suppose that an application allows embedded font information to be transmitted by means of the 8-bit sequence

$$\text{^ATimes^B} = 01,54,69,6D,65,73,02.$$

Then the corresponding sequence of Unicode character codes would be

$$\text{^ATimes^B} = 0001,0054,0069,006D,0065,0073,0002$$

where the embedded text is viewed as Unicode text, or

$$\text{^ATimes^B} = 0001,5469,6D65,7300,0002$$

where the embedded data is interpreted by some other protocol. It cannot be

$$\text{^ATimes^B} = 0154,696D,6573,0200$$

because, in a Unicode character encoding, this sequence represents three characters—U+0154 LATIN CAPITAL LETTER R ACUTE, and the two Han characters U+696D and U+6573—and a fourth value, U+0200, is currently unassigned. None of these is a control character. If a control sequence contains embedded binary data, then that data does not need to be zero-extended because the control sequence constitutes a higher protocol.

If a system does use control code sequences to embed additional information, then such sequences form a higher-level protocol. Such higher-level protocols are not specified by the Unicode standard; their existence cannot be assumed without a separate agreement.

For example, use of the ISO control sequences (extended to 16 bits) for controlling bidirectional formatting is a legitimate higher-level protocol layered on top of the plain text of the Unicode encoding.

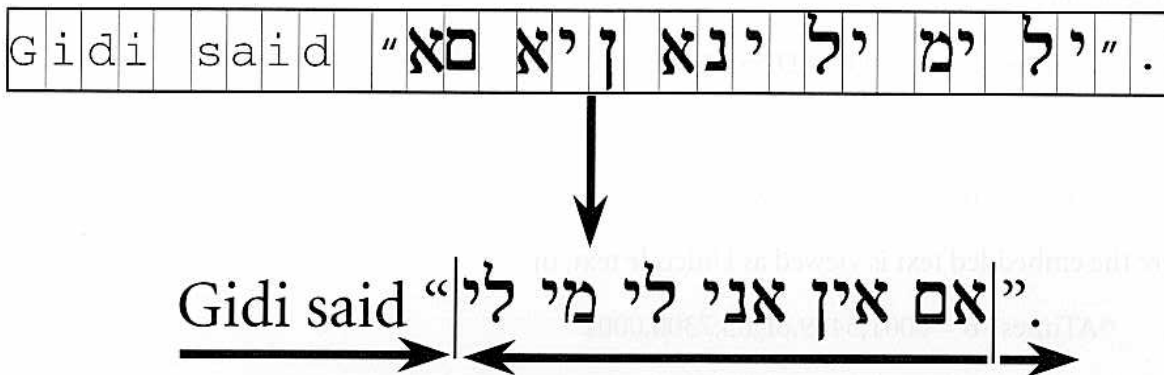
### *Paragraph/Line Separators*

Implementers may follow existing practices in the use of control characters for these designators. The Unicode standard provides unambiguous characters, U+2028 LINE SEPARATOR, and U+2029 PARAGRAPH SEPARATOR, for general use. This is the canonical form of paragraph and line separation in the Unicode encoding.

### *Ordering of Character Sequences*

In a Unicode encoding, text is stored in sequential order in the backing store. Logical or backing store order corresponds to the order in which text is typed on the keyboard after corrections such as insertions, deletions, and overtyping have taken place. Conversion of Unicode text in the backing store to readable text is handled by higher-level text rendering processes.

The distinction between logical order and display order for reading is shown in the following figure:



*Figure 2-2. Bidirectional Ordering*

In the examples, the first character of each backing store string is at position 0, but in the displayed text, logical order is independent of display order. When text is ordered for display, the glyph that represents the character at position 0 of the English text is at the left. The logical start character of the Hebrew text, however, is represented by the glyph closest to the right margin. The succeeding Hebrew glyphs are laid out to the left.

In a Unicode encoding, all scripts are stored in logical order. This applies even when characters of different dominant direction are mixed: left-to-right (Greek, Roman, Thai) with right-to-left (Arabic, Hebrew), or with vertical (Mongolian) script. Properties of directionality inherent in most characters determine the correct display order of text. But an author will sometimes override

the inherent directionality for literary reasons. Moreover, some characters do not have inherent directionality (such as spaces and punctuation). Therefore, the Unicode encoding scheme includes characters to specify changes in direction. Appendix A provides rules for the correct presentation of text containing left-to-right and right-to-left scripts. Characters such as the medial form of the *short i* in Devanagari are displayed before the characters that they logically follow in the backing store. (See the Devanagari character block description for further explanation.)

Non-spacing marks (accent marks in the Greek and Roman scripts, vowel marks in Arabic and Devanagari, and so on) do not appear linearly in the final rendered text. In a Unicode character code sequence, all such characters follow the base character which they modify, or the character after which they would be articulated in phonetic order (for example, Roman “ã” is stored as “a~” when not stored in a precomposed form).

### *Alternate Spellings*

In many cases, the same graphic appearance can be produced by several different sequences of Unicode values. These cases can result where there is a precomposed character that is the same as a composed character sequence, when there are two non-spacing marks whose order does not determine a different placement, or when there are alternate characters with different reordering semantics. For example:

à

LATIN SMALL LETTER A + NON-SPACING DOT ABOVE

ã

LATIN SMALL LETTER A TILDE

LATIN SMALL LETTER A + NON-SPACING TILDE

â

LATIN SMALL LETTER A + NON-SPACING DOT BELOW + NON-SPACING DOT ABOVE

LATIN SMALL LETTER A + NON-SPACING DOT ABOVE + NON-SPACING DOT BELOW

ã

LATIN SMALL LETTER A TILDE + NON-SPACING DOT BELOW

LATIN SMALL LETTER A + NON-SPACING TILDE + NON-SPACING DOT BELOW

LATIN SMALL LETTER A + NON-SPACING DOT BELOW + NON-SPACING TILDE

Similarly, the appearance is the same for the sequences THAI VOWEL SIGN SARA E + THAI LETTER KO KAI and THAI LETTER KO KAI + THAI PHONETIC ORDER VOWEL SIGN SARA E.

In such cases, the Unicode standard does not prescribe one particular sequence; all of the sequences are equivalent. Systems may choose to normalize Unicode text to one particular sequence, such as normalizing composed character sequences into precomposed characters or vice versa. Only a relatively small number of precomposed base-character-plus-diacritics have independent Unicode character values, namely those which have existed in dominant standards.

## 2.5 Non-spacing Marks

Characters intended to be positioned relative to an associated base character are depicted in the character code charts above, below, or through a dotted circle. They are also annotated in the names list or in the character properties list as “non-spacing.” When rendered, these characters are intended to be positioned relative to the preceding base character in some manner, and not to occupy a spacing position by themselves. This is the motivation for the term “non-spacing.” Diacritics are the principal class of non-spacing marks used in European alphabets. (In the charts for Indian scripts, some vowels are depicted to the left of dotted circles. This is a special case to be carefully distinguished from that of general non-spacing characters. Such vowel signs are rendered to the left of a consonant letter or consonant cluster, even though their logical order in the Unicode encoding is following the consonant letter. The decision to code these in pronunciation order and not in visual order was made by the developers of the ISCII standard.)

There is a separate block for generic diacritics, intended to be used with any script. There is an additional block for symbol diacritics, intended to be used with symbol base characters. There are additional non-spacing marks in the blocks for particular scripts when they are primarily used with these scripts. However, the allocation of a non-spacing mark to one block or another identifies only its primary usage; it is not intended to define or limit the range of characters to which it may be applied. In the Unicode standard, all sequences of character codes are permitted.

Some scripts, such as Hebrew, Arabic, and the scripts of India and Southeast Asia, also have non-spacing marks indicated in the charts in relation to dotted circles to show their position relative to the base character. Many of these non-spacing marks encode vowel letters; as such they are not generally referred to as “diacritics.”

Typical diacritics have a very strong interaction with the base character to which they are applied, in the sense that the combination is a semantically indivisible unit. However, in the Unicode standard, the term “diacritic” is interpreted more broadly, to include accents as well.



### *Sequence of Base Letters and Non-Spacing Marks*

In the Unicode standard, all non-spacing marks are encoded following the base characters. The Unicode sequence U+0061 LATIN SMALL LETTER A “a” + U+0308 NON-SPACING DIAERESIS “¨” + U+0075 LATIN SMALL LETTER U “u” unambiguously encodes as “äü” not “aü.”

This convention is different from the convention in ISO 6937, and the bibliographic standard ISO 5926. The reasons for the old convention were conformity with “dead keys” on mechanical typewriters (no longer a consideration for computers), and considerations of efficiency in serial parsing of character streams which included diacritics.

The convention used by the Unicode standard is consistent with the logical order of other non-spacing marks in Semitic and Indic scripts, the great majority of which follow the base characters with respect to which they are positioned. To avoid the complication of defining and implementing non-spacing marks on both sides of base characters, the Unicode standard specifies that all non-spacing marks must follow their base characters. This convention conforms to the way modern font technology handles the rendering of non-spacing graphical forms, so that mapping from character store to font rendering is simplified.

### *Spacing Clones of European Diacritical Marks*

By convention, diacritical marks used by the Unicode encoding scheme may be exhibited in (apparent) isolation by applying them to U+0020 SPACE or to U+00A0 NON-BREAKING SPACE. This might be done, for example, when talking about the diacritical mark itself as a mark, rather than using it in its normal way in text. The Unicode standard separately encodes clones of many common European diacritical marks that are spacing characters, largely to provide compatibility with existing character sets. These related characters are cross-referenced.

### *Multiple Non-spacing Marks*

There are instances where more than one diacritical mark is applied to a single base character. The Unicode standard does not restrict the number of non-spacing marks that can follow a baseform character. The following rules apply:

1. If the non-spacing marks can interact typographically—for example, a non-spacing macron and a non-spacing diaeresis—then the order of graphic display is determined by the order of coded characters. The diacritics or other non-spacing marks are positioned from the base character outward. Non-spacing marks placed above a base character will be stacked vertically, starting with the first encountered in the logical store and continuing for as many marks above as are required by the character codes following the base character. For non-spacing marks placed below a base character, the situation is inverted, with the non-spacing marks starting from the base character and stacking downward.

An example of multiple non-spacing characters above the base character is found in Thai, where a consonant letter can have above it one of the vowels U+0E34 through U+0E37 and, above that, one of four tone marks U+0E48 through U+0E4B. The order of character codes which produces this graphic display is base consonant character code, plus vowel character code, plus one tone character code.

2. Some specific non-spacing marks override the default stacking behavior by being positioned horizontally rather than stacking, or by ligaturing with an adjacent non-spacing mark. When positioned horizontally, the order of codes is reflected by positioning in the dominant order of the script with which they are used. For example, horizontal accents, in a left-to-right script would be coded left-to-right.

Prominent characters that show such override behavior are associated with specific scripts or alphabets. The Greek “breathing marks” U+0371 and U+0372 require that they together with a following acute or grave accent be rendered side-by-side above a letter, rather than the accent marks being stacked above the breathing marks. The order of codes here is base character code plus breathing mark code plus accent mark code.



GREEK SMALL LETTER ALPHA + GREEK NON-SPACING PSILI PNEUMATA + NON-SPACING ACUTE



GREEK SMALL LETTER ALPHA + NON-SPACING ACUTE + GREEK NON-SPACING PSILI PNEUMATA

Two Vietnamese tone marks which have the same graphic appearance as the Latin acute and grave accent marks do not stack above the three Vietnamese vowel letters which already contain the circumflex diacritic (â ê ô). Instead, they form ligatures with the circumflex component of the base vowel characters.



LATIN SMALL LETTER A CIRCUMFLEX + NON-SPACING ACUTE TONE MARK

â

LATIN SMALL LETTER A CIRCUMFLEX + NON-SPACING ACUTE

LATIN SMALL LETTER A + NON-SPACING CIRCUMFLEX + NON-SPACING ACUTE

â

LATIN SMALL LETTER A ACUTE + NON-SPACING CIRCUMFLEX

LATIN SMALL LETTER A + NON-SPACING ACUTE + NON-SPACING CIRCUMFLEX

3. If the non-spacing marks cannot interact typographically—for example, when one nonspacing mark is above a baseform and another is below—then the Unicode standard does not specify that a distinct graphic form will be rendered if the codes are in different orders. In such cases it is up to the user and application to decide on a consistent and useful order of codes. (This may often reflect the fact that one non-spacing character is more tightly bound structurally to the base letter than the other.) No distinction of graphic form will generally result from such alternative orderings of codes.

## 2.6 Formal Requirements for Unicode Conformance

A computing system (a hardware device or a software application) may perform various processes on text character sequences. This clause specifies in general terms when a text process conforms to the Unicode standard. The purpose of this clause is to state clearly the intentions of the Unicode design in order to provide implementors a method of producing systems that conform to the Unicode standard and that will work consistently and cooperatively with each other. This clause is not intended to prescribe a procedure for evaluating systems as conforming or non-conforming.

**IN SUMMARY:** For each 16-bit character code in a text sequence, a process that conforms to the Unicode standard must either interpret the code value according to its Unicode semantics as specified in this standard, or not interpret it at all. An interchange process may not change a code that it cannot interpret; other than that, the behavior of processes relative to codes it cannot interpret is unspecified.

### 1. *Public Interchange of bit sequences as character codes*

“Interchange” refers to processes which transmit and receive (including store and retrieve) sequence of bits that are to be treated as sequences of coded text characters. “Public Interchange” refers to processes that exchange or record bits in such a way that the bits might ever be accessed by other processes that conform to the Unicode encoding scheme, and which are not under their control or coordination (for example, transmission in e-mail or storage to a file). “Private Interchange” refers to processes which exchange or record bits in such a way that access to those bits is

available only to other processes (not necessarily Unicode-conformant) under their control or coordination (for example, encrypted transmission or storage in a temporary data structure).

The terms “Unicode” and “Unicode text” refer exclusively to the unique representation of character code sequences specified above. This design is not intended to preclude encrypted, compressed, or byte-swapped text from being used in Private Interchange settings, but it does separate such usage from the question of public conformance to the Unicode standard.

The basic conformance requirements for Public Interchange of character codes are:

*A conformant process must treat textual information in 16-bit units, and must obey the rules specified elsewhere in this standard regarding each sub-range of Unicode values:*

U+0000 → U+001F	Control codes
U+0020 → U+007E	Graphic character codes
U+007F → U+009F	Control codes
U+00A0 → U+E7FF	Graphic character codes
U+E800 → U+FDFF	Private Use Area
U+FE00 → U+FEFE	Compatibility Zone
U+FEFF	Byte order mark
U+FF00 → U+FFEF	Compatibility Zone
U+FFF0 → U+FFFD	Graphic character codes
U+FFFE → U+FFFF	Not character codes
Bidirectional text	Appendix A

*A Unicode-conformant process must not assign any semantic or character identity to any code point left unassigned in this version of the standard, with the exception of the codes in the Private Use Area.*

NOTE: This does not preclude the assignment of certain generic semantics (for example, left-to-right or right-to-left directionality) which allow graceful behavior of algorithms in the presence of codes which are outside the adopted subset.

Unicode code points are 16-bit quantities. Machine architectures differ in the ordering of whether the most significant byte or the least significant byte comes first. These are known as “big-endian” and “little-endian” orders, respectively.

The Unicode standard does not specify any order of bytes or bits inside the 16-bit sequence of a Unicode code point. However, in Public Interchange and in the absence of any information to the contrary provided by a higher protocol, a conformant process may assume that Unicode character sequences it receives are in the order of the most significant byte first.



NOTE: The majority of all Interchange occurs with processes running on the same or a similar configuration. This makes intra-domain Interchange of Unicode text in the domain-specific byte order fully conformant, and limits the role of the canonical byte order to Interchange of Unicode text across domain, or where the nature of the originating domain is unknown. Processes may prefix data with U+FEFF BYTE ORDER MARK, and a receiving process may interpret that character as verification that the text arrived with the byte order expected by the receiving process. Alternatively, on receiving U+FFFE, the receiving process may recover text data by attempting to re-read it in byte-swapped order.

## 2. Interpretation

“Interpretation” refers to processes which take 16-bit values as input and produce results based on the assumption that these values represent codes for specific text characters with known identities (semantics). The conformance requirement with regard to Interpretation is:

*If a conformant process is able to Interpret a given character code, then the Interpretation must be consistent with the Unicode character semantics.*

One important usage of Interpretation merits particular mention. “Rendering” (or “Presentation”) refers to processes having access to fonts and other resources, which take 16-bit values as input, interpret their character identities, and draw a visible graphic depiction of the text. As a corollary of the preceding, the conformance requirement with regard to Rendering is:

*If a conformant Rendering process is able to Interpret and draw a given character code, then the graphic depiction must be consistent with the Unicode character semantics.*

## 3. Modification

“Modification” refers to processes which make any change at all to a sequence of bits that are to be treated as sequences of text characters. The conformance requirement with regard to Modification is:

*A conformant (Public) Interchange process which in any way receives and retransmits encoded text must not change the 16-bit value of any character code that it cannot interpret. It must transmit such a code unchanged from the value that was received.*

In other words, any change that a conformant system makes in a given sequence of bits treated as text characters must be made intentionally by processes that have knowingly interpreted the codes.

#### 4. Interpretable subsets of characters

In conforming to the Unicode standard, no conditions are set regarding the subset of 16-bit values that any process may or may not be able to Interpret. As a corollary of all the foregoing, the conformance requirement with regard to interpretable subsets is:

*A conformant process may or may not be able to Interpret (including Render) any particular set of 16-bit values. In general, the behavior of a conformant process with respect to code values that it cannot Interpret is intentionally left unspecified. However, if a process cannot Interpret a character code and is required to Interchange it, it must transmit the code unchanged.*

This design is not intended to preclude enumerations or specifications of the characters that a process or system is able to Interpret, but it does separate supported subset enumerations from the question of conformance. In real life, any system may occasionally receive an unfamiliar character code which it is unable to Interpret; all that really matters is that it be able to retransmit the code undamaged.

#### *Examples of Conformant and Non-Conformant Processing*

The following are examples of text processes that conform to the Unicode standard:

- A system receives any text sequence and retransmits it unchanged. Whether or not it could have performed any other process on the text (display it, spell-check it, and so on) is immaterial.
- A system receives a sequence of English text and retransmits it all converted to uppercase (presumably an intentional change consistent with the text's semantics).
- A system is capable of rendering only characters in the "8859-1" range (Unicode values 0000-00FF), although it treats those code numbers in full 16-bit form. Such a system may be unable to render other character codes or sequences legibly (for example, if it is given a sequence of Bengali characters). It might render a black box or ring a bell or do nothing. Behavior in such cases is not specified or restricted.
- A system renders a given sequence of English text in any Latin font style, line length, page layout and so on of its choice, such that the text is conventionally readable with the intended interpretation.

The following are examples of text processes that do not conform to the Unicode standard:

- A system receives a sequence of English text and retransmits it all converted to random Bengali characters, or vice versa (presumably an unintentional change inconsistent with the text's semantics).

- A system takes in a sequence of Unicode characters and treats it as though it were a sequence of ASCII bytes.
- A system produces results constituting an Interpretation of an unassigned Unicode value (excluding User Space).
- A system interprets non-spacing marks as preceding base characters. For example, the sequence “a”, “¨”, “u” as “äü” and not “äu.”

