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

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

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

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

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

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

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

QA268.U545 2004
005.7'2—dc21
2003052158

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

Chapter 2

General Structure

This chapter discusses the fundamental principles governing the design of the Unicode Standard and presents an informal overview of its main features. The chapter starts by placing the Unicode Standard in an architectural context by discussing the nature of text representation and text processing and its bearing on character encoding decisions. Next, the Unicode Design Principles are introduced—ten basic principles that convey the essence of the standard. The Unicode Design Principles serve as a kind of tutorial framework for understanding the Unicode Standard, and they are a useful place from which to start to get a summary of the overall nature of the standard.

The chapter then moves on to the Unicode character encoding model, introducing the concepts of character, code point, and encoding forms, and diagramming the relationships between them. This provides an explanation of the encoding forms UTF-8, UTF-16, and UTF-32 and some general guidelines regarding the circumstances under which one form would be preferable to another.

The section on Unicode allocation then describes the overall structure of the Unicode codespace, showing a summary of the code charts and the locations of blocks of characters associated with different scripts or sets of symbols.

Next, the chapter discusses the issue of writing direction and introduces several special types of characters important for understanding the Unicode Standard. In particular, the use of *combining* characters, the *byte order mark*, and *control* characters is explored in some detail.

Finally, there is an informal statement of the conformance requirements for the Unicode Standard. This informal statement, with a number of easy-to-understand examples, gives a general sense of what conformance to the Unicode Standard means. The rigorous, formal definition of conformance is given in the subsequent *Chapter 3, Conformance*.

2.1 Architectural Context

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, because these directly serve the needs of a system's users. Character codes are like nuts and bolts—minor, but essential and ubiquitous components used in many different ways in the construction of computer software systems. No single design of a character set can be optimal for all uses, so the architecture of the Unicode Standard strikes a balance among several competing requirements.

Basic Text Processes

Most computer systems provide low-level functionality for a small number of basic text processes from which more sophisticated text-processing capabilities are built. The following text processes are supported by most computer systems to some degree:

- Rendering characters visible (including ligatures, contextual forms, and so on)
- Breaking lines while rendering (including hyphenation)
- Modifying appearance, such as point size, kerning, underlining, slant, and weight (light, demi, bold, and so on)
- Determining units such as “word” and “sentence”
- Interacting with users in processes such as selecting and highlighting text
- Accepting keyboard input and editing stored text through insertion and deletion
- Comparing text in operations such as determining the sort order of two strings, or filtering or matching strings
- Analyzing text content in operations such as spell-checking, hyphenation, and parsing morphology (that is, determining word roots, stems, and affixes)
- Treating text as bulk data for operations such as compressing and decompressing, truncating, transmitting, and receiving

Text Elements, Characters, and Text Processes

One of the more profound challenges in designing a worldwide character encoding stems from the fact that, for each text process, written languages differ in what is considered a fundamental unit of text, or a *text element*.

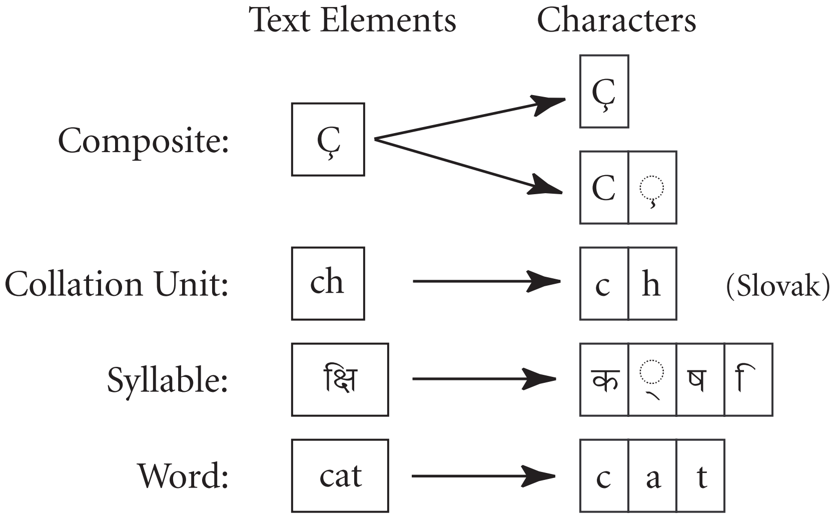
For example, in traditional German orthography, the letter combination “ck” is a text element for the process of hyphenation (where it appears as “k-k”), but not for the process of sorting; in Spanish, the combination “ll” may be a text element for the traditional process of sorting (where it is sorted between “l” and “m”), but not for the process of rendering; and in English, the letters “A” and “a” are usually distinct text elements for the process of rendering, but generally not distinct for the process of searching text. The text elements in a given language depend upon the specific text process; a text element for spell-checking may have different boundaries from a text element for sorting purposes. For example, in the phrase “the quick brown fox”, the sequence “fox” is a text element for the purpose of spell-checking.

However, a character encoding standard provides just the fundamental units of encoding (that is, the abstract characters), which must exist in a unique relationship to the assigned numerical *code points*. Assigned characters are the smallest interpretable units of stored text.

Figure 2-1 illustrates the relationship between several different types of text elements and the characters that are used to represent those text elements. Unicode Standard Annex #29, “Text Boundaries,” provides more details regarding the specifications of boundaries for such text elements as user-perceived characters (called *grapheme clusters*), words, and sentences.

The design of the character encoding must provide precisely the set of characters that allows programmers to design applications capable of implementing a variety of text pro-

Figure 2-1. Text Elements and Characters



cesses in the desired languages. These characters may not map directly to any particular set of text elements that is used by one of these processes.

Text Processes and Encoding

In the case of English text using an encoding scheme such as ASCII, the relationships between the encoding and the basic text processes built on it are seemingly straightforward: characters are generally rendered visible one by one in distinct rectangles from left to right in 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 such as the Unicode Standard, the relationship between the encoding and implementation of basic text processes must be considered explicitly, for several reasons:

- Many assumptions about character rendering that hold true for the English alphabet fail for other writing systems. Characters in these other writing systems 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. See *Section 8.2, Arabic*, and *Section 9.1, Devanagari*, for detailed examples of this situation.
- It is not always obvious that one set of text characters is an optimal encoding for a given language. For example, two approaches exist for the encoding of accented characters commonly used in French or Swedish: ISO/IEC 8859 defines letters such as "ä" and "ö" as individual characters, whereas ISO 5426 represents them by composition with diacritics instead. In the Swedish language, both are considered distinct letters of the alphabet, following the letter "z". In French, the diaeresis on a vowel merely marks it as being pronounced in isolation. In practice, both approaches can be used to implement either language.
- No encoding can support all basic text processes equally well. As a result, some trade-offs are necessary. For example, ASCII defines separate codes for uppercase and lowercase letters. This choice causes some text processes, such as rendering, to be carried out more easily, but other processes, such as comparison,

> to become more difficult. A different encoding design for English, such as case-shift control codes, would have the opposite effect. In designing a new encoding scheme for complex scripts, such trade-offs must be evaluated and decisions made explicitly, rather than unconsciously.

For these reasons, design of the Unicode Standard is not specific to the design of particular basic text-processing algorithms. Instead, it provides an encoding that can be used with a wide variety of algorithms. 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. Culturally expected sorting orders require arbitrarily complex sorting algorithms. The expected sort sequence for the same characters differs across languages; thus, in general, no single acceptable lexicographic ordering exists. See Unicode Technical Standard #10, "Unicode Collation Algorithm," for the standard default mechanism for comparing Unicode strings.

Text processes supporting many languages are often more complex than they are for English. The character encoding design of the Unicode Standard strives to minimize this additional complexity, enabling modern computer systems to interchange, render, and manipulate text in a user's own script and language—and possibly in other languages as well.

2.2 Unicode Design Principles

The design of the Unicode Standard reflects the 10 fundamental principles stated in *Table 2-1*. Not all of these principles can be satisfied simultaneously. The design strikes a balance between maintaining consistency for the sake of simplicity and efficiency and maintaining compatibility for interchange with existing standards.

Table 2-1. The 10 Unicode Design Principles

Principle	Statement
Universality	The Unicode Standard provides a single, universal repertoire.
Efficiency	Unicode text is simple to parse and process.
Characters, not glyphs	The Unicode Standard encodes characters, not glyphs.
Semantics	Characters have well-defined semantics.
Plain text	Unicode characters represent plain text.
Logical order	The default for memory representation is logical order.
Unification	The Unicode Standard unifies duplicate characters within scripts across languages.
Dynamic composition	Accented forms can be dynamically composed.
Equivalent sequences	Static precomposed forms have an equivalent dynamically composed sequence of characters.
Convertibility	Accurate convertibility is guaranteed between the Unicode Standard and other widely accepted standards.

Universality

The Unicode Standard encodes a single, very large set of characters, encompassing all the characters needed for worldwide use. This single repertoire is intended to be universal in coverage, containing all the characters for textual representation in all modern writing systems, in most historic writing systems for which sufficient information is available to enable reliable encoding of characters, and symbols used in plain text.

Because the universal repertoire is known and well defined in the standard, it is possible to specify a rich set of character semantics. By relying on those character semantics, implementations can provide detailed support for complex operations on text at a reasonable cost.

The Unicode Standard, by supplying a universal repertoire associated with well-defined character semantics, does not require the *code set independent* model of internationalization and text handling. That model abstracts away string handling as manipulation of byte streams of unknown semantics to protect implementations from the details of hundreds of different character encodings, and selectively late-binds locale-specific character properties to characters. Of course, it is always possible for code set independent implementations to retain their model and to treat Unicode characters as just another character set in that context. It is not at all unusual for Unix implementations to simply add UTF-8 as another character set, parallel to all the other character sets they support. However, by contrast, the Unicode approach—because it is associated with a universal repertoire—assumes that characters and their properties are inherently and inextricably associated. If an internationalized application can be structured to work directly in terms of Unicode characters, all levels of the implementation can reliably and efficiently access character storage and be assured of the universal applicability of character property semantics.

Efficiency

The Unicode Standard is designed to make efficient implementation possible. There are no escape characters or shift states in the Unicode character encoding model. Each character code has the same status as any other character code; all codes are equally accessible.

All Unicode encoding forms are self-synchronizing and non-overlapping. This makes randomly accessing and searching inside streams of characters efficient.

By convention, characters of a script are grouped together as far as is practical. Not only is this practice convenient for looking up characters in the code charts, but it makes implementations more compact and compression methods more efficient. The common punctuation characters are shared.

Formatting characters are given specific and unambiguous functions in the Unicode Standard. This design simplifies the support of subsets. To keep implementations simple and efficient, stateful controls and formatting characters are avoided wherever possible.

Characters, Not Glyphs

The Unicode Standard draws a distinction between *characters* and *glyphs*. Characters are the abstract representations of the smallest components of written language that have semantic value. They represent primarily, but not exclusively, the letters, punctuation, and other signs that constitute natural language text and technical notation. Characters are represented by code points that reside only in a memory representation, as strings in memory, or on disk. The Unicode Standard deals only with character codes.

Glyphs represent the shapes that characters can have when they are rendered or displayed. In contrast to characters, glyphs appear on the screen or paper as particular representations of one or more characters. A repertoire of glyphs makes up a font. Glyph shape and methods of identifying and selecting glyphs are the responsibility of individual font vendors and of appropriate standards and are not part of the Unicode Standard.

Various relationships may exist 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. The distinction between characters and glyphs is illustrated in *Figure 2-2*.

Figure 2-2. Characters Versus Glyphs

Glyphs	Unicode Characters
A A *A* 𝒜 𝔄 A *A* 𝒜 A	U+0041 LATIN CAPITAL LETTER A
a a *a* 𝒶 𝔞 a *a* *a* **a**	U+0061 LATIN SMALL LETTER A
ﬁ fi	U+0066 LATIN SMALL LETTER F + U+0069 LATIN SMALL LETTER I
п *п* *п̄*	U+043F CYRILLIC SMALL LETTER PE
ه ـه هـ ـهـ	U+0647 ARABIC LETTER HEH

Even the letter “a” has a wide variety of glyphs that can represent it. A lowercase Cyrillic “п” also has a variety of glyphs; the second glyph shown in *Figure 2-2* is customary for italic in Russia, while the third is customary for italic in Serbia. Sequences such as “fi” may be shown with two independent glyphs or with a ligature glyph. Arabic letters are shown with different glyphs, depending on their position in a word; the glyphs in *Figure 2-2* show independent, final, initial, and medial forms.

For certain scripts, such as Arabic and the various Indic scripts, the number of glyphs needed to display a given script may be significantly larger than the number of characters encoding the basic units of that script. The number of glyphs may also depend on the orthographic style supported by the font. For example, an Arabic font intended to support the *Nastaliq* style of Arabic script may possess many thousands of glyphs. However, the character encoding employs the same few dozen letters regardless of the font style used to depict the character data in context.

A font and its associated rendering process define an arbitrary mapping from Unicode characters to glyphs. Some of the glyphs in a font may be independent forms for individual characters; others may be rendering forms that do not directly correspond to any single character.

The process of mapping from characters in the memory representation to glyphs is one aspect of text rendering. The final appearance of rendered text may also depend on context (neighboring characters in the memory representation), 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 prototypical shape of a letter or character, as shown in *Figure 2-3*.

For the Latin script, this relationship between character code sequence and glyph is relatively simple and well known; for several other scripts, it is documented in this standard. However, in all cases, fine typography requires a more elaborate set of rules than given here. The Unicode Standard documents the default relationship between character sequences and glyphic appearance for the purpose of ensuring that the same text content can be stored with the same, and therefore interchangeable, sequence of character codes.

What the user thinks of as a single character—which may or may not be represented by a single glyph—may be represented in the Unicode standard as multiple code points. See *Figure 2-4* for examples.

Figure 2-3. Unicode Character Code to Rendered Glyphs

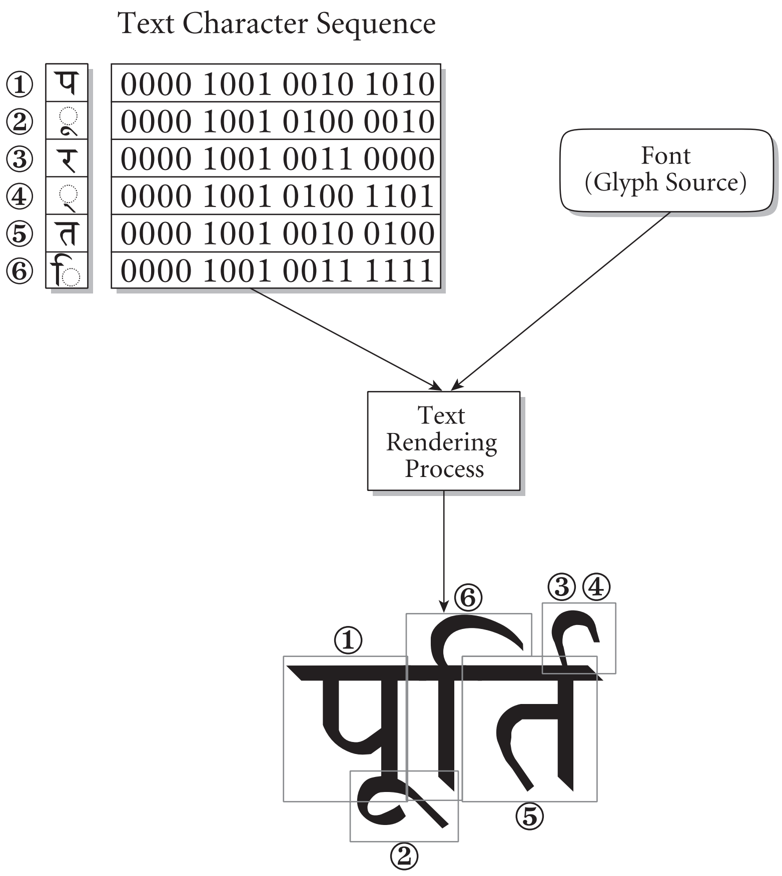


Figure 2-4. User Characters as Multiple Code Points

Character	Code Points	Linguistic Usage
ch	0063 0068	Slovak, traditional Spanish
tʰ	0074 02B0	Native American languages
x̣	0078 0323	
ƛ̓	019B 0313	
ą́	00E1 0328	Lithuanian
i̇́	0069 0307 0301	
ト゚	30C8 309A	Ainu (in kana transcription)

Semantics

Characters have well-defined semantics. Character property 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 numeric, spacing, combination, and directionality properties (see *Chapter 4, Character Properties*). Additional properties may be defined as needed from time to time. By itself, neither the character name nor its location in the code table designates its properties.

Plain Text

Plain text is a pure sequence of character codes; plain Unicode-encoded text is therefore a sequence of Unicode character codes. In contrast, *styled text*, also known as *rich text*, is any text representation consisting of plain text plus added information such as a language identifier, font size, color, hypertext links, and so on. For example, the text of this book, a multifont text as formatted by a desktop publishing system, is rich text.

The simplicity of plain text gives it a natural role as a major structural element of rich text. SGML, RTF, HTML, XML, and TEX are examples of rich text fully represented as plain text streams, interspersing plain text data with sequences of characters that represent the additional data structures. They use special conventions embedded within the plain text file, such as "<p>", to distinguish the markup or *tags* from the "real" content. Many popular word processing packages rely on a buffer of plain text to represent the content, and implement links to a parallel store of formatting data.

The relative functional roles of both plain and rich text are well established:

- Plain text is the underlying content stream to which formatting can be applied.
- Rich text carries complex formatting information as well as text context.
- Plain text is public, standardized, and universally readable.
- Rich text representation may be implementation-specific or proprietary.

Although some rich text formats have been standardized or made public, the majority of rich text designs are vehicles for particular implementations and are not necessarily readable by other implementations. Given that rich text equals plain text plus added information, the extra information in rich text can always be stripped away to reveal the "pure" text underneath. This operation is often employed, for example, in word processing systems that use both their own private rich text format and plain text file format as a universal, if limited, means of exchange. Thus, by default, plain text represents the basic, interchangeable content of text.

Plain text represents character content only, not its appearance. It can be displayed in a variety of ways and requires a rendering process to make it visible with a particular appearance. 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. Instead, the disparate rendering processes are simply required to make the text legible according to the intended reading. This legibility criterion constrains the range of possible appearances. The relationship between appearance and content of plain text may be summarized as follows:

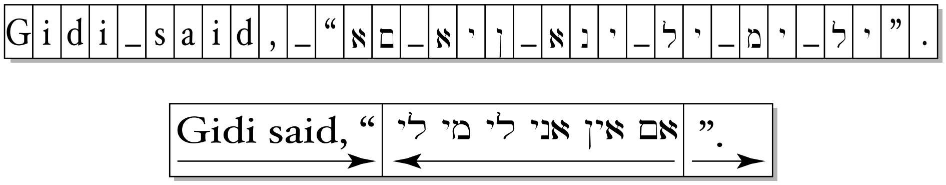
> *Plain text must contain enough information to permit the text to be rendered legibly, and nothing more.*

The Unicode Standard encodes plain text. The distinction between plain text and other forms of data in the same data stream is the function of a higher-level protocol and is not specified by the Unicode Standard itself.

Logical Order

Unicode text is stored in *logical order* in the memory representation, roughly corresponding to the order in which text is typed in via the keyboard. In some circumstances, the order of characters differs from this logical order when the text is displayed or printed. Where needed to ensure consistent legibility, the Unicode Standard defines the conversion of Unicode text from the memory representation to readable (displayed) text. The distinction between logical order and display order for reading is shown in *Figure 2-5*.

Figure 2-5. Bidirectional Ordering


G i d i _ s a i d , _ “ א ם _ א י ן _ א נ י _ ל י _ מ י _ ל י ” .
Gidi said, “
אם אין אני לי מי לי
”.

When the text in *Figure 2-5* is ordered for display, the glyph that represents the first character of the English text appears at the left. The logical start character of the Hebrew text, however, is represented by the Hebrew glyph closest to the right margin. The succeeding Hebrew glyphs are laid out to the left.

Logical order applies even when characters of different dominant direction are mixed: left-to-right (Greek, Cyrillic, Latin) with right-to-left (Arabic, Hebrew), or with vertical script. Properties of directionality inherent in characters generally determine the correct display order of text. The Unicode bidirectional algorithm specifies how these properties are used to resolve directional interactions when characters of right-to-left and left-to-right directionality are mixed. (See Unicode Standard Annex #9, “The Bidirectional Algorithm.”) However, this inherent directionality is occasionally insufficient to render plain text legibly. This can occur in certain situations when characters of different directionality are mixed. The Unicode Standard therefore includes characters to specify changes in direction for use when the inherent directionality of characters is insufficient. The bidirectional algorithm provides rules that use these directional layout control characters together with the inherent directional properties of characters to provide the correct presentation of text containing both left-to-right and right-to-left scripts. This allows for exact control of the display ordering for legible interchange and also ensures that plain text used for simple items like file names or labels can always be correctly ordered for display.

For the most part, logical order corresponds to *phonetic order*. The only current exceptions are the Thai and Lao scripts, which employ visual ordering; in these two scripts, users traditionally type in visual order rather than phonetic order.

Characters such as the *short i* in Devanagari are displayed before the characters that they logically follow in the memory representation. (See *Section 9.1, Devanagari,* for further explanation.)

Combining marks (accent marks in the Greek, Cyrillic, and Latin scripts, vowel marks in Arabic and Devanagari, and so on) do not appear linearly in the final rendered text. In a Unicode character sequence, all such characters *follow* the base character that they modify. For example, the Latin letter “x̣” is stored as “x” followed by combining “◌̣”.

Unification

The Unicode Standard avoids duplicate encoding of characters by unifying them within scripts across languages; characters that are equivalent 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. (See *Section 11.1, Han.*)

It is quite normal for many characters to have different usages, such as *comma* "," for either thousands-separator (English) or decimal-separator (French). The Unicode Standard avoids duplication of characters due to specific usage in different languages; rather, it duplicates characters *only* to support compatibility with base standards. Avoidance of duplicate encoding of characters is important to avoid visual ambiguity.

There are a few notable instances in the standard where visual ambiguity between different characters is tolerated, however. For example, in most fonts there is little or no distinction visible between Latin "o", Cyrillic "o", and Greek "o" (*omicron*). These are not unified because they are characters from three different scripts, and there are many legacy character encodings that distinguish them. As another example, there are three characters whose glyph is the same uppercase barred D shape, but they correspond to three distinct lowercase forms. Unifying these uppercase characters would have resulted in unnecessary complications for case mapping.

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: just as French *i grec*, German *ypsilon*, and English *wye* are all represented by the same character code, U+0057 "Y", so too are Chinese *zi*, Japanese *ji,* and Korean *ja* all represented as the same character code, U+5B57 字.

In determining whether to unify variant ideograph forms across standards, the Unicode Standard follows the principles described in *Section 11.1, Han*. Where these principles determine that two forms constitute a trivial difference, the Unicode Standard assigns a single code. Otherwise, separate codes are assigned.

There are many characters in the Unicode Standard that could have been unified with existing visually similar Unicode characters, or that could have been omitted in favor of some other Unicode mechanism for maintaining the kinds of text distinctions for which they were intended. However, considerations of interoperability with other standards and systems often require that such compatibility characters be included in the Unicode Standard. The status of a character as a compatibility character does not mean that the character is deprecated in the standard.

Dynamic Composition

The Unicode Standard allows for the dynamic composition of accented forms and Hangul syllables. Combining characters used to create composite forms are productive. Because the process of character composition is open-ended, new forms with modifying marks may be created from a combination of base characters followed by combining characters. For example, the diaeresis, "¨", may be combined with all vowels and a number of consonants in languages using the Latin script and several other scripts.

Equivalent Sequences

Some text elements can be encoded either as static precomposed forms or by dynamic composition. Common precomposed forms such as U+00DC "Ü" LATIN CAPITAL LETTER U WITH DIAERESIS are included for compatibility with current standards. For static precomposed forms, the standard provides a mapping to an equivalent dynamically composed sequence of characters. (See also *Section 3.7, Decomposition.*) Thus, different sequences of Unicode characters are considered equivalent. For example, a precomposed character may be represented as a composed character sequence (see *Figure 2-6* and *Figure 2-18*).

In cases involving two or more sequences considered to be equivalent, the Unicode Standard does not prescribe one particular sequence as being the *correct* one; instead, each

Figure 2-6. Equivalent Sequences

B + Ä → B + A + ◌̈

LJ + A → L + J + A

sequence is merely equivalent to the others. In *Figure 2-6*, the sequences on each side of the arrows express the same content and would be interpreted the same way.

If an application or user attempts to distinguish non-identical sequences which are nonetheless considered to be equivalent sequences, as shown in the examples in *Figure 2-6*, it would not be guaranteed that other applications or users would recognize the same distinctions. To prevent introducing interoperability problems between applications, such distinctions must be avoided wherever possible.

Where a unique representation is required, a normalized form of Unicode text can be used to eliminate unwanted distinctions. The Unicode Standard defines four normalization forms: Normalization Form D (NFD), Normalization Form KD (NFKD), Normalization Form C (NFC), and Normalization Form KC (NFKC). Roughly speaking, NFD and NFKD decompose characters where possible, while NFC and NFKC compose characters where possible. For more information, see Unicode Standard Annex #15, "Unicode Normalization Forms," and *Section 5.6, Normalization*.

Decompositions. Precomposed characters are formally known as decomposables, because they have decompositions to one or more *other* characters. There are two types of decompositions:

- *Canonical.* The character and its decomposition should be treated as essentially equivalent.
- *Compatibility.* The decomposition may remove some information (typically formatting information) that is important to preserve in particular contexts. By definition, compatibility decomposition is a superset of canonical decomposition.

Thus there are three types of characters, based on their decomposition behavior:

- *Canonical decomposable.* The character has a distinct canonical decomposition.
- *Compatibility decomposable.* The character has a distinct compatibility decomposition.
- *Nondecomposable.* The character has no distinct decomposition, neither canonical nor compatibility. Loosely speaking, these characters are said to have "no decomposition," even though technically they decompose to themselves.

Figure 2-7 illustrates these three types.

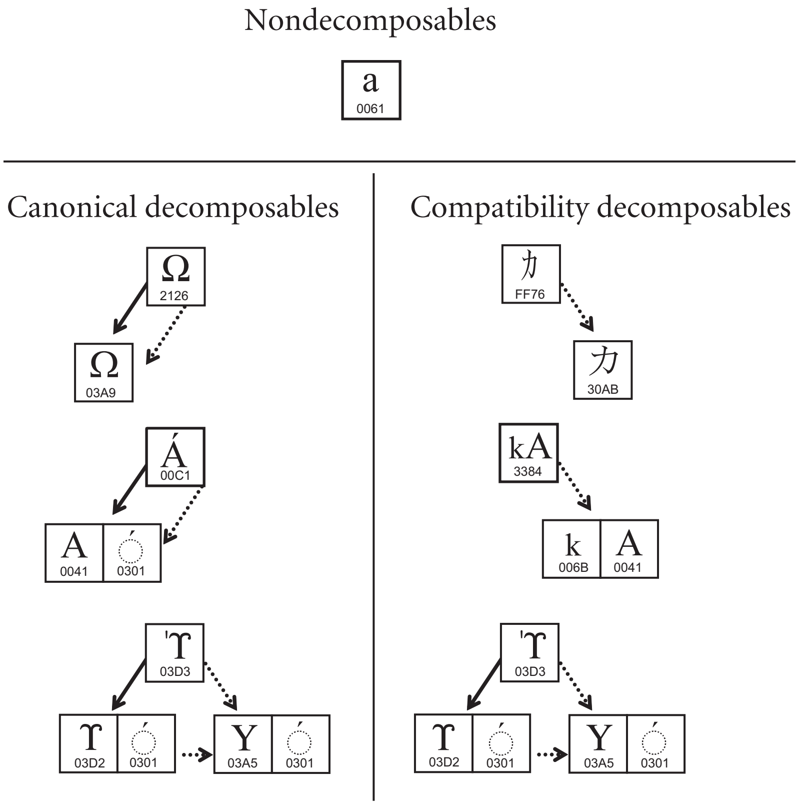
The solid arrows in *Figure 2-7* indicate canonical decompositions, and the dotted arrows indicate compatibility decompositions. The figure illustrates two important points:

- Decompositions may be to single characters *or* to sequences of characters. Decompositions to a single character, also known as *singleton decompositions*, are seen for the *ohm sign* and the *halfwidth katakana ka* in the figure. Because of examples like these, decomposable characters in Unicode do not always consist of obvious, separate parts; one can only know their status by examining the data tables for the standard.

- There are a very small number of characters that are both canonical and compatibility decomposable. The example shown in the figure is for the Greek hooked upsilon symbol with an acute accent. It has a canonical decomposition to one sequence and a compatibility decomposition to a different sequence.

For more precise definitions of some of these terms, see *Chapter 3, Conformance*.

Figure 2-7. Types of Decomposables



Convertibility

Character identity is preserved for interchange with a number of different base standards, including national, international, and vendor standards. Where variant forms (or even the same form) are given separate codes within one base standard, they are also kept separate within the Unicode Standard. This choice guarantees the existence of a mapping between the Unicode Standard and base standards.

Accurate convertibility is guaranteed between the Unicode Standard and other standards in wide usage as of May 1993. In general, a single code point in another standard will correspond to a single code point in the Unicode Standard. Sometimes, however, a single code point in another standard corresponds to a sequence of code points in the Unicode Standard, or vice versa. Conversion between Unicode text and text in other character codes must in general be done by explicit table-mapping processes. (See also *Section 5.1, Transcoding to Other Standards.*)

2.3 Compatibility Characters

Compatibility Characters

Compatibility characters are those that would not have been encoded except for compatibility and round-trip convertibility with other standards. They are variants of characters that already have encodings as *normal* (that is, non-compatibility) characters in the Unicode Standard. Examples of compatibility characters in this sense include all of the glyph variants in the Compatibility and Specials Area: halfwidth or fullwidth characters from East Asian character encoding standards, Arabic contextual form glyphs from preexisting Arabic code pages, Arabic ligatures and ligatures from other scripts, and so on. Other examples include CJK compatibility ideographs, which are generally duplicates of a unified Han ideograph, legacy alternate format characters such as U+206C INHIBIT ARABIC FORM SHAPING, and fixed-width space characters used in old typographical systems.

The Compatibility and Specials Area contains a large number of compatibility characters, but the Unicode Standard also contains many compatibility characters that do not appear in that area. These include examples such as U+2163 "IV" ROMAN NUMERAL FOUR, U+2007 FIGURE SPACE, and U+00B2 "^{2}" SUPERSCRIPT TWO.

Compatibility Decomposable Characters

There is a second, narrow sense of "compatibility character" in the Unicode Standard, corresponding to the notion of *compatibility decomposable* introduced in *Section 2.2, Unicode Design Principles*. This sense of compatibility character is strictly defined as any Unicode character whose compatibility decomposition is not identical to its canonical decomposition. (See definition D21 in *Section 3.7, Decomposition*.) Because a compatibility character in this narrow sense must also be a composite character, it may also be unambiguously referred to as a compatibility composite character, or *compatibility composite* for short.

In the past, compatibility decomposable characters have simply been ambiguously referred to as compatibility characters. This has occasionally led implementers astray, because it could easily result in assuming that all compatibility characters have decomposition mappings, for example. The terminological distinction has been introduced to help prevent implementers from making such mistakes.

The implementation implications for compatibility decomposable characters are different than those for compatibility characters in general. The compatibility decomposable characters are precisely defined in the Unicode Character Database, whereas the compatibility characters in the more inclusive sense are not. It is important to remember that not all compatibility characters are compatibility decomposables. For example, the deprecated alternate format characters do not have any distinct decomposition, and CJK compatibility ideographs have *canonical* decomposition mappings rather than compatibility decomposition mappings.

Mapping Compatibility Characters

Identifying one character as a compatibility variant of another character usually implies that the first can be remapped to the other without the loss of any textual information other than formatting and layout. However, such remapping cannot always take place because many of the compatibility characters are included in the standard precisely to allow systems to maintain one-to-one mappings to other existing character encoding standards and code pages. In such cases, a remapping would lose information that is important to maintaining some distinction in the original encoding. By definition, a compatibility decomposable character decomposes into a compatibly equivalent character or character sequence. Even in such cases, an implementation must proceed with due caution—replacing one with the other may change not only formatting information, but also other textual distinctions on which some other process may depend.

In many cases there exists a visual relationship between a compatibility composition and a standard character that is akin to a font style or directionality difference. Replacing such characters with unstyled characters could affect the meaning of the text. Replacing them with rich text would preserve the meaning for a human reader, but could cause some programs that depend on the distinction to behave unpredictably.

2.4 Code Points and Characters

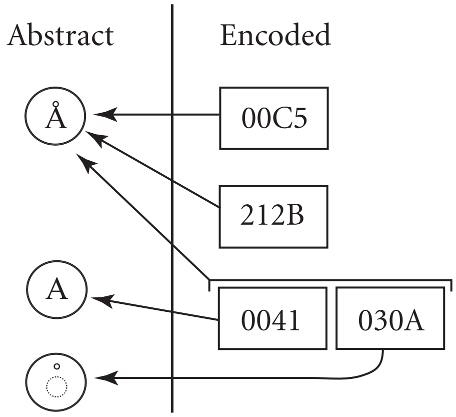
On a computer, abstract characters are encoded internally as numbers. To create a complete character encoding, it is necessary to define the list of all characters to be encoded and to establish systematic rules for how the numbers represent the characters.

The range of integers used to code the abstract characters is called the *codespace*. A particular integer in this set is called a *code point*. When an abstract character is mapped or *assigned* to a particular code point in the codespace, it is then referred to as an *encoded character*.

In the Unicode Standard, the codespace consists of the integers from 0 to $10FFFF_{16}$, comprising 1,114,112 code points available for assigning the repertoire of abstract characters. Of course, there are constraints on how the codespace is organized, and particular areas of the codespace have been set aside for encoding of certain kinds of abstract characters or for other uses in the standard. For more on the *allocation* of the Unicode codespace, see *Section 2.8, Unicode Allocation*.

Figure 2-8 illustrates the relationship between abstract characters and code points, which together constitute encoded characters. Note that some abstract characters may be associated with multiple, separately encoded characters (that is, be encoded “twice”). In other instances, an abstract character may be represented by a sequence of two (or more) other encoded characters. The solid arrows connect encoded characters with the abstract characters that they represent and encode.

Figure 2-8. Codespace and Encoded Characters



When referring to code points in the Unicode Standard, the usual practice is to refer to them by their numeric value expressed in hexadecimal, with a "U+" prefix. (See *Section 0.3, Notational Conventions.*) Encoded characters can also be referred to by their code points only, but to prevent ambiguity, the official Unicode name of the character is often also added; this clearly identifies the abstract character that is encoded. For example:

U+0061 LATIN SMALL LETTER A

U+10330 GOTHIC LETTER AHSA

U+201DF CJK UNIFIED IDEOGRAPH-201DF

Such citations refer only to the encoded character per se, associating the code point (as an integral value) with the abstract character that is encoded.

Types of Code Points

There are many different ways to categorize code points. *Table 2-2* illustrates some of the categorizations and basic terminology used in the Unicode Standard.

Table 2-2. Types of Code Points

Basic Type	Brief Description	General Category	Character Status	Code Point Status
Graphic	Letter, mark, number, punctuation, symbol, and spaces	L, M, N, P, S, Zs	*Assigned to abstract character*	*Designated (assigned) code point*
Format	Invisible but affects neighboring characters; includes line/paragraph separators	Cf, Zl, Zp		
Control	Usage defined by protocols or standards outside the Unicode Standard	Cc		
Private-use	Usage defined by private agreement outside the Unicode Standard	Co		
Surrogate	Permanently reserved for UTF-16; restricted interchange	Cs	*Not assigned to abstract character*	
Noncharacter	Permanently reserved for internal usage; restricted interchange	Cn		
Reserved	Reserved for future assignment; restricted interchange			*Undesignated (unassigned) code point*

Not all assigned code points represent abstract characters; only Graphic, Format, Control and Private-use do. Surrogates and Noncharacters are assigned code points but not assigned to abstract characters. Reserved code points are assignable: any may be assigned in a future version of the standard. The General Category provides a finer breakdown of Graphic characters, and is also used to distinguish the other basic types (except between Noncharacter and Reserved). Other properties defined in the Unicode Character Database provide for different categorizations of Unicode code points.

Control Codes. Sixty-five code points (U+0000..U+001F and U+007F..U+009F) are reserved specifically as control codes, for compatibility with the C0 and C1 control codes of the ISO/IEC 2022 framework. A few of these control codes are given specific interpretations by the Unicode Standard. (See *Section 15.1, Control Codes.*)

Noncharacters. Sixty-six code points are not used to encode characters. Noncharacters consist of U+FDD0..U+FDEF and the last two code points on each plane, including U+FFFE and U+FFFF on the BMP. (See *Section 15.8, Noncharacters.*)

Private Use. Three ranges of code points have been set aside for private use. Characters in these areas will never be defined by the Unicode Standard. These code points can be freely used for characters of any purpose, but successful interchange requires an agreement between sender and receiver on their interpretation. (See *Section 15.7, Private-Use Characters.*)

Surrogates. 2,048 code points have been allocated for surrogates, which are used in the UTF-16 encoding form. (See *Section 15.5, Surrogates Area.*)

Restricted Interchange. Code points that are not assigned to abstract characters are subject to restrictions in interchange.

- Surrogate code points cannot be conformantly interchanged using Unicode encoding forms. They do not correspond to Unicode scalar values, and thus do not have well-formed representations in any Unicode encoding form.
- Noncharacter code points are reserved for internal use, such as for sentinel values. They should never be interchanged. They do, however, have well-formed representations in Unicode encoding forms and survive conversions between encoding forms. This allows sentinel values to be preserved internally across Unicode encoding forms, even though they are not designed to be used in open interchange.
- All implementations need to preserve reserved code points because they may originate in implementations that use a *future* version of the Unicode Standard. For example, suppose that one person is using a Unicode 4.0 system and a second person is using a Unicode 3.2 system. The first person sends the second person a document containing some code points newly assigned in Unicode 4.0; these code points were unassigned in Unicode 3.2. The second person may edit the document, not changing the reserved codes, and send it on. In that case the second person is interchanging what are, as far as the second person knows, reserved code points.

Code Point Semantics. The semantics of most code points are established by this standard; the exceptions are Controls, Private-use, and Noncharacters. Control codes generally have semantics determined by other standards or protocols (such as ISO/IEC 6429), but there are a small number of control codes for which the Unicode Standard specifies particular semantics. See *Table 15-1* in *Section 15.1, Control Codes*, for the exact list of those control codes. The semantics of private-use characters are outside the scope of the Unicode Standard; their use is determined by private agreement, as, for example, between vendors. Noncharacters have semantics in internal use only.

2.5 Encoding Forms

Computers handle numbers not simply as abstract mathematical objects, but as combinations of fixed-size units like bytes and 32-bit words. A character encoding model must take this fact into account when determining how to associate numbers with the characters.

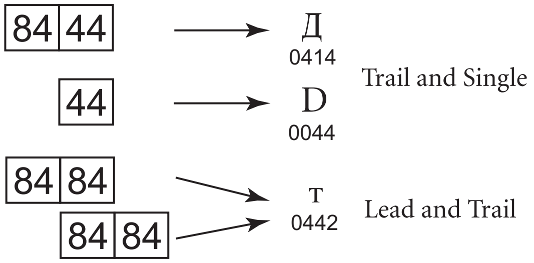
Actual implementations in computer systems represent integers in specific *code units* of particular size—usually 8-bit (= byte), 16-bit, or 32-bit. In the Unicode character encoding model, precisely defined *encoding forms* specify how each integer (code point) for a Unicode character is to be expressed as a sequence of one or more code units. The Unicode Standard provides three distinct encoding forms for Unicode characters, using 8-bit, 16-bit, and 32-bit units. These are correspondingly named UTF-8, UTF-16, and UTF-32. (The "UTF" is a carryover from earlier terminology meaning Unicode (or UCS) Transformation Format.) Each of these three encoding forms is an equally legitimate mechanism for representing Unicode characters; each has advantages in different environments.

All three encoding forms can be used to represent the full range of encoded characters in the Unicode Standard; they are thus fully interoperable for implementations that may choose different encoding forms for various reasons. Each of the three Unicode encoding forms can be efficiently transformed into either of the other two without any loss of data.

Non-overlap. Each of the Unicode encoding forms is designed with the principle of non-overlap in mind. This means that if a given code point is represented by a certain sequence of one or more code units, it is impossible for any other code point to ever be represented by the same sequence of code units.

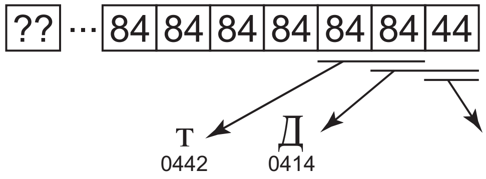
To illustrate the problems with overlapping encodings, see *Figure 2-9*. In this encoding (Windows code page 932), characters are formed from either one or two code bytes. Whether a sequence is one or two in length depends on the first byte, so that the values for lead bytes (of a two-byte sequence) and single bytes are disjoint. However, single-byte values and trail-byte values can overlap. That means that when someone searches for the character "D", for example, they might find it (mistakenly) as the trail byte of a two-byte sequence, or as a single, independent byte. To find out which alternative is correct, a program must look backward through text.

Figure 2-9. Overlap in Legacy Mixed-Width Encodings



The situation is made more complex by the fact that lead and trail bytes can also overlap, as in the second part of *Figure 2-9*. This means that the backward scan has to repeat until it hits the start of the text or hits a sequence that could not exist as a pair as shown in *Figure 2-10*. This is not only inefficient, it is extremely error-prone: corruption of one byte can cause entire lines of text to be corrupted.

Figure 2-10. Boundaries and Interpretation



The Unicode encoding forms avoid this problem, because *none* of the ranges of values for the lead, trail, or single code units in any of those encoding forms overlap.

Non-overlap makes all of the Unicode encoding forms well behaved for searching and comparison. When searching for a particular character, there will never be a mismatch against some code unit sequence that represents just part of another character. The fact that all Unicode encoding forms observe this principle of non-overlap distinguishes them from many legacy East Asian multibyte character encodings, for which overlap of code unit sequences may be a significant problem for implementations.

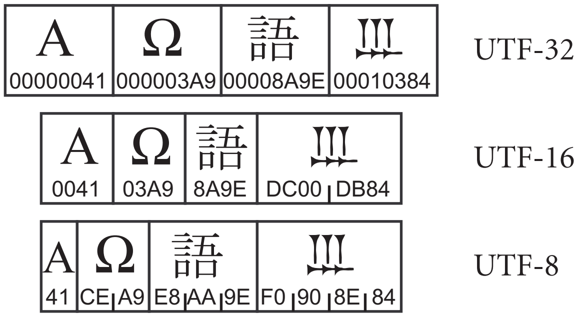
Another aspect of non-overlap in the Unicode encoding forms is that all Unicode characters have determinate boundaries when expressed in any of the encoding forms. That is, the edges of code unit sequences representing a character are easily determined by local examination of code units; there is never any need to scan back indefinitely in Unicode text to correctly determine a character boundary. This property of the encoding forms has sometimes been referred to as *self-synchronization*. This property has another very important implication: corruption of a single code unit corrupts *only* a single character; none of the surrounding characters are affected.

For example, when randomly accessing a string, a program can find the boundary of a character with limited backup. In UTF-16, if a pointer points to a leading surrogate, a single backup is required. In UTF-8, if a pointer points to a byte starting with 10xxxxxx (in binary), one to three backups are required to find the beginning of the character.

Conformance. The Unicode Consortium fully endorses the use of any of the three Unicode encoding forms as a conformant way of implementing the Unicode Standard. It is important not to fall into the trap of trying to distinguish "UTF-8 *versus* Unicode," for example. UTF-8, UTF-16, and UTF-32 are *all* equally valid and conformant ways of implementing the encoded characters of the Unicode Standard.

Figure 2-11 shows the three Unicode encoding forms, including how they are related to Unicode code points.

Figure 2-11. Unicode Encoding Forms



In *Figure 2-11*, the UTF-32 line shows that each example character can be expressed with one 32-bit code unit. Those code units have the same values as the code point for the character. For UTF-16, most characters can be expressed with one 16-bit code unit, whose value is the same as the code point for the character, but characters with high code point values require a pair of 16-bit surrogate code units instead. In UTF-8, a character may be expressed with one, two, three, or four bytes, and the relationship between those byte values and the code point value is more complex.

UTF-8, UTF-16, and UTF-32 are further described in the subsections that follow. See each subsection for a general overview of how each encoding form is structured and the general benefits or drawbacks of each encoding form for particular purposes. For the detailed formal definition of the encoding forms and conformance requirements, see *Section 3.9, Unicode Encoding Forms*.

UTF-32

UTF-32 is the simplest Unicode encoding form. Each Unicode code point is represented directly by a single 32-bit code unit. Because of this, UTF-32 has a one-to-one relationship between encoded character and code unit; it is a fixed-width character encoding form. This makes UTF-32 an ideal form for APIs that pass single character values.

As for all of the Unicode encoding forms, UTF-32 is restricted to representation of code points in the range $0..10FFFF_{16}$—that is, the Unicode codespace. This guarantees interoperability with the UTF-16 and UTF-8 encoding forms.

The value of each UTF-32 code unit corresponds exactly to the Unicode code point value. This situation differs significantly from that for UTF-16 and especially UTF-8, where the code unit values often change unrecognizably from the code point value. For example, U+10000 is represented as <00010000> in UTF-32, but it is represented as <F0 90 80 80> in UTF-8. For UTF-32 it is trivial to determine a Unicode character from its UTF-32 code unit representation, whereas UTF-16 and UTF-8 representations often require doing a code unit conversion before the character can be identified in the Unicode code charts.

UTF-32 may be a preferred encoding form where memory or disk storage space for characters is no particular concern, but where fixed-width, single code unit access to characters is desired. UTF-32 is also a preferred encoding form for processing characters on most Unix platforms.

UTF-16

In the UTF-16 encoding form, code points in the range U+0000..U+FFFF are represented as a single 16-bit code unit; code points in the supplementary planes, in the range U+10000..U+10FFFF, are instead represented as pairs of 16-bit code units. These pairs of special code units are known as *surrogate pairs*. The values of the code units used for surrogate pairs are completely disjunct from the code units used for the single code unit representations, thus maintaining non-overlap for all code point representations in UTF-16. For the formal definition of surrogates, see *Section 3.8, Surrogates*.

UTF-16 optimizes the representation of characters in the Basic Multilingual Plane (BMP)—that is, the range U+0000..U+FFFF. For that range, which contains the vast majority of common-use characters for all modern scripts of the world, each character requires only one 16-bit code unit, thus requiring just half the memory or storage of the UTF-32 encoding form. For the BMP, UTF-16 can effectively be treated as if it were a fixed-width encoding form.

However, for supplementary characters, UTF-16 requires two 16-bit code units. The distinction between characters represented with one versus two 16-bit code units means that formally UTF-16 is a variable-width encoding form. That fact can create implementation difficulties, if not carefully taken into account; UTF-16 is somewhat more complicated to handle than UTF-32.

UTF-16 may be a preferred encoding form in many environments that need to balance efficient access to characters with economical use of storage. It is reasonably compact, and all the common, heavily used characters fit into a single 16-bit code unit.

UTF-16 is the historical descendant of the earliest form of Unicode, which was originally designed to use a fixed-width, 16-bit encoding form exclusively. The surrogates were added to provide an encoding form for the supplementary characters at code points past U+FFFF. The design of the surrogates made them a simple and efficient extension mechanism that works well with older Unicode implementations, and that avoids many of the problems of other variable-width character encodings. See *Section 5.4, Handling Surrogate Pairs in UTF-16*, for more information about surrogates and their processing.

For the purpose of sorting text, note that binary order for data represented in the UTF-16 encoding form is not the same as code point order. This means that a slightly different comparison implementation is needed for code point order. For more information, see *Section 5.17, Binary Order*.

UTF-8

To meet the requirements of byte-oriented, ASCII-based systems, a third encoding form is specified by the Unicode Standard: UTF-8. It is a variable-width encoding form that preserves ASCII transparency, making use of 8-bit code units.

Much existing software and practice in information technology has long depended on character data being represented as a sequence of bytes. Furthermore, many of the protocols depend not only on ASCII values being invariant, but must make use of or avoid special byte values that may have associated control functions. The easiest way to adapt Unicode implementations to such a situation is to make use of an encoding form that is already defined in terms of 8-bit code units and that represents all Unicode characters while not disturbing or reusing any ASCII or C0 control code value. That is the function of UTF-8.

UTF-8 is a variable-width encoding form, using 8-bit code units, in which the high bits of each code unit indicate the part of the code unit sequence to which each byte belongs. A range of 8-bit code unit values is reserved for the first, or *leading,* element of a UTF-8 code unit sequences, and a completely disjunct range of 8-bit code unit values is reserved for the subsequent, or *trailing,* elements of such sequences; this convention preserves non-overlap for UTF-8. *Table 3-5* on page 77 shows how the bits in a Unicode code point are distributed among the bytes in the UTF-8 encoding form. See *Section 3.9, Unicode Encoding Forms*, for the full, formal definition of UTF-8.

The UTF-8 encoding form maintains transparency for all of the ASCII code points (0x00..0x7F). That means Unicode code points U+0000..U+007F are converted to single bytes 0x00..0x7F in UTF-8, and are thus indistinguishable from ASCII itself. Furthermore, the values 0x00..0x7F do not appear in any byte for the representation of any other Unicode code point, so that there can be no ambiguity. Beyond the ASCII range of Unicode, many of the non-ideographic scripts are represented by two bytes per code point in UTF-8; all non-surrogate code points between U+0800 and U+FFFF are represented by three bytes; and supplementary code points above U+FFFF require four bytes.

UTF-8 is typically the preferred encoding form for HTML and similar protocols, particularly for the Internet. The ASCII transparency helps migration. UTF-8 also has the advantage that it is already inherently byte-serialized, as for most existing 8-bit character sets; strings of UTF-8 work easily with C or other programming languages, and many existing APIs that work for typical Asian multibyte character sets adapt to UTF-8 as well with little or no change required.

In environments where 8-bit character processing is required for one reason or another, UTF-8 also has the following attractive features as compared to other multibyte encodings:

- The first byte of a UTF-8 code unit sequence indicates the number of bytes to follow in a multibyte sequence. This allows for very efficient forward parsing.

- It is also efficient to find the start of a character when beginning from an arbitrary location in a byte stream of UTF-8. Programs need to search at most four bytes backward, and usually much less. It is a simple task to recognize an initial byte, because initial bytes are constrained to a fixed range of values.
- As with the other encoding forms, there is no overlap of byte values.

Comparison of the Advantages of UTF-32, UTF-16, and UTF-8

On the face of it, UTF-32 would seem to be the obvious choice of Unicode encoding forms for an internal processing code because it is a fixed-width encoding form. It can be conformantly bound to the C and C++ wchar_t, which means that such programming languages may offer built-in support and ready-made string APIs that programmers can take advantage of. However, UTF-16 has many countervailing advantages that may lead implementers to choose it instead as an internal processing code.

While all three encoding forms need at most 4 bytes (or 32 bits) of data for each character, in practice UTF-32 in almost all cases for real data sets occupies twice the storage that UTF-16 requires. Therefore, a common strategy is to have internal string storage use UTF-16 or UTF-8, but to use UTF-32 when manipulating individual characters.

On average, more than 99 percent of all UTF-16 data is expressed using single code units. This includes nearly all of the typical characters that software needs to handle with special operations on text—for example, format control characters. As a consequence, most text scanning operations do not need to unpack UTF-16 surrogate pairs at all, but can safely treat them as an opaque part of a character string.

For many operations, UTF-16 is as easy to handle as UTF-32, and the performance of UTF-16 as a processing code tends to be quite good. UTF-16 is the internal processing code of choice for a majority of implementations supporting Unicode. Other than for Unix platforms, UTF-16 provides the right mix of compact size with the ability to handle the occasional character outside the BMP.

UTF-32 has somewhat of an advantage when it comes to simplicity of software coding design and maintenance. Because the character handling is fixed-width, UTF-32 processing does not require maintaining branches in the software to test and process the double code unit elements required for supplementary characters by UTF-16. On the other hand, 32-bit indices into large tables are not particularly memory efficient. To avoid the large memory penalties of such indices, Unicode tables are often handled as multistage tables (see “Multistage Tables” in *Section 5.1, Transcoding to Other Standards*). In such cases, the 32-bit code point values are sliced into smaller ranges for segmented access to the tables. This is true even in typical UTF-32 implementations.

The performance of UTF-32 as a processing code may actually be worse than UTF-16 for the same data, because the additional memory overhead means that cache limits will be exceeded more often and memory paging will occur more frequently. For systems with processor designs that have penalties for 16-bit aligned access, but with very large memories, this effect may be less.

In any event, Unicode code points do *not* necessarily match user expectations for “characters.” For example, the following are not represented by a single code point: a combining character sequence such as <g, acute>; a conjoining jamo sequence for Korean; or the Devanagari conjunct “ksha.” Because some Unicode text processing must be aware of and handle such sequences of characters as text elements, the fixed-width encoding form advantage of UTF-32 is somewhat offset by the inherently variable-width nature of processing text elements. See Unicode Technical Report #18, “Unicode Regular Expression

Guidelines," for an example where commonly implemented processes deal with inherently variable-width text elements, owing to user expectations of the identity of a "character."

UTF-8 is reasonably compact in terms of the number of bytes used. It is really only at a significant size disadvantage when used for East Asian implementations such as Chinese, Japanese, and Korean, which use Han ideographs or Hangul syllables requiring three-byte code unit sequences in UTF-8. UTF-8 is also significantly less efficient in processing than the other encoding forms.

A binary sort of UTF-8 strings gives the same ordering as a binary sort of Unicode code points. This is also, obviously, the same order as for a binary sort of UTF-32 strings.

All three encoding forms give the same results for binary string comparisons or string sorting when dealing only with BMP characters (in the range U+0000..U+FFFF). However, when dealing with supplementary characters (in the range U+10000..U+10FFFF), UTF-16 binary order does not match Unicode code point order. This can lead to complications when trying to interoperate with binary sorted lists—for example, between UTF-16 systems and UTF-8 or UTF-32 systems. However, for data that is sorted according to the conventions of a specific language or locale, rather than using binary order, data will be ordered the same, regardless of the encoding form.

2.6 Encoding Schemes

The discussion of Unicode encoding forms in the previous section was concerned with the machine representation of Unicode code units. Each code unit is represented in a computer simply as a numeric data type; just as for other numeric types, the exact way the bits are laid out internally is irrelevant to most processing. However, interchange of textual data, particularly between computers of different architectural types, requires consideration of the exact ordering of the bits and bytes involved in numeric representation. Integral data, including character data, is *serialized* for open interchange into well-defined sequences of bytes. This process of *byte serialization* allows all applications to correctly interpret exchanged data and to accurately reconstruct numeric values (and thereby character values) from it. In the Unicode Standard, the specifications of the distinct types of byte serializations to be used with Unicode data are known as Unicode *encoding schemes*.

Modern computer architectures differ in *ordering* in terms of whether the most significant byte or the least significant byte of a large numeric data type comes first in internal representation. These sequences are known as "big-endian" and "little-endian" orders, respectively. For the Unicode 16- and 32-bit encoding forms (UTF-16 and UTF-32), the specification of a byte serialization must take into account the big-endian or little-endian architecture of the system on which the data is represented, so that when the data is byte-serialized for interchange it will be well defined.

A *character encoding scheme* consists of a specified character encoding form plus a specification of how the code units are serialized into bytes. The Unicode Standard also specifies the use of an initial *byte order mark* (BOM) to explicitly differentiate big-endian or little-endian data in some of the Unicode encoding schemes. (See the "Byte Order Mark" subsection in *Section 15.9, Specials*.)

When a higher-level protocol supplies mechanisms for handling the endianness of integral data types, it is not necessary to use Unicode encoding schemes or the byte order mark. In those cases Unicode text is simply a sequence of integral data types.

For UTF-8, the encoding scheme consists merely of the UTF-8 code units (= bytes) in sequence. Hence, there is no issue of big- versus little-endian byte order for data represented in UTF-8. However, for 16-bit and 32-bit encoding forms, byte serialization must break up the code units into two or four bytes, respectively, and the order of those bytes must be clearly defined. Because of this, and because of the rules for the use of the byte order mark, the three encoding forms of the Unicode Standard result in a total of seven Unicode encoding schemes, as shown in *Table 2-3*.

Table 2-3. The Seven Unicode Encoding Schemes

Encoding Scheme	Endian Order	BOM Allowed?
UTF-8	N/A	yes
UTF-16	Big-endian or Little-endian	yes
UTF-16BE	Big-endian	no
UTF-16LE	Little-endian	no
UTF-32	Big-endian or Little-endian	yes
UTF-32BE	Big-endian	no
UTF-32LE	Little-endian	no

The endian order entry for UTF-8 in *Table 2-3* is marked N/A because UTF-8 code units are 8 bits in size, and the usual machine issues of endian order for larger code units do not apply. The serialized order of the bytes must not depart from the order defined by the UTF-8 encoding form. Use of a BOM is neither required nor recommended for UTF-8, but may be encountered in contexts where UTF-8 data is converted from other encoding forms that use a BOM, or where the BOM is used as a UTF-8 signature. See the “Byte Order Mark” subsection in *Section 15.9, Specials*, for more information.

Note that some of the Unicode encoding schemes have the same labels as the three Unicode encoding forms. This could cause confusion, so it is important to keep the context clear when using these terms: character encoding *forms* refer to integral data units in memory or in APIs, and byte order is irrelevant; character encoding *schemes* refer to byte-serialized data, as for streaming I/O or in file storage, and byte order *must* be specified or determinable.

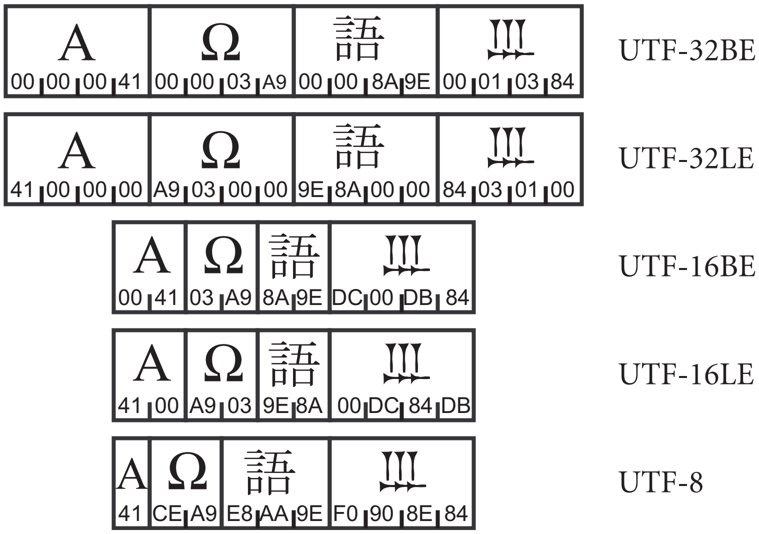
The Internet Assigned Names Authority (IANA) maintains a registry of *charset names* used on the Internet. Those charset names are very close in meaning to the Unicode character encoding model’s concept of character encoding schemes, and all of the Unicode character encoding schemes are in fact registered as *charsets*. While the two concepts are quite close, and the names used are identical, some important differences may arise in terms of the requirements for each, particularly when it comes to handling of the byte order mark. Exercise due caution when equating the two.

Figure 2-12 illustrates the Unicode character encoding schemes, showing how each is derived from one of the encoding forms by serialization of bytes.

In *Figure 2-12*, the code units used to express each example character have been serialized into sequences of bytes. This figure should be compared with *Figure 2-11*, which shows the same characters before serialization into sequences of bytes. The “BE” lines show serialization in big-endian order, whereas the “LE” lines show the bytes reversed into little-endian order. For UTF-8, the code unit is just an 8-bit byte, so that there is no distinction between big-endian and little-endian order. UTF-32 and UTF-16 encoding schemes using the *byte order mark* are not shown in *Figure 2-12*, to keep the basic picture regarding serialization of bytes clearer.

For the detailed formal definition of the Unicode encoding schemes and conformance requirements, see *Section 3.10, Unicode Encoding Schemes*. For further general discussion about character encoding forms and character encoding schemes, both for the Unicode Standard and as applied to other character encoding standards, see Unicode Technical Report #17, “Character Encoding Model.” For information about charsets and character

Figure 2-12. Unicode Encoding Schemes


A Ω 語 𐌄
00 00 00 41 00 00 03 A9 00 00 8A 9E 00 01 03 84
UTF-32BE
41 00 00 00 A9 03 00 00 9E 8A 00 00 84 03 01 00
UTF-32LE
00 41 03 A9 8A 9E DC 00 DB 84
UTF-16BE
41 00 A9 03 9E 8A 00 DC 84 DB
UTF-16LE
41 CE A9 E8 AA 9E F0 90 8E 84
UTF-8

conversion, see Unicode Technical Report #22, "Character Mapping Markup Language (CharMapML)."

2.7 Unicode Strings

A Unicode string datatype is simply an ordered sequence of code units. Thus a Unicode 8-bit string is an ordered sequence of 8-bit code units, a Unicode 16-bit string is an ordered sequence of 16-bit code units, and a Unicode 32-bit string is an ordered sequence of 32-bit code units.

Depending on the programming environment, a Unicode string may or may not also be required to be in the corresponding Unicode encoding form. For example, strings in Java, C#, or ECMAScript are Unicode 16-bit strings, but are not necessarily well-formed UTF-16 sequences. In normal processing, it can be far more efficient to allow such strings to contain code unit sequences that are not well-formed UTF-16—that is, isolated surrogates. Because strings are such a fundamental component of every program, checking for isolated surrogates in every operation that modifies strings can be significant overhead, especially because supplementary characters are extremely rare as a percentage of overall text in programs worldwide.

It is straightforward to design basic string manipulation libraries that handle isolated surrogates in a consistent and straightforward manner. They cannot ever be interpreted as abstract characters, but can be internally handled the same way as noncharacters where they occur. Typically they occur only ephemerally, such as in dealing with keyboard events. While an ideal protocol would allow keyboard events to contain complete strings, many allow only a single UTF-16 code unit per event. As a sequence of events is transmitted to the application, a string that is being built up by the application in response to those events may contain isolated surrogates at any particular point in time.

However, whenever such strings are specified to be in a particular Unicode encoding form—even one with the same code unit size—the string must not violate the requirements of that encoding form. For example, isolated surrogates in a Unicode 16-bit string are not allowed when that string is specified to be *well-formed* UTF-16. (See *Section 3.9,* *Unicode Encoding Forms*.) There are a number of techniques for dealing with an isolated surrogate, such as omitting it, or converting it into U+FFFD REPLACEMENT CHARACTER to produce well-formed UTF-16, or simply halting the processing of the string with an error. For more information on this topic, see Unicode Technical Report #22, "Character Mapping Markup Language (CharMapML)."

2.8 Unicode Allocation

For convenience, the encoded characters of the Unicode Standard are grouped by linguistic and functional categories, such as script or writing system. For practical reasons, there are occasional departures from this general principle, as when punctuation associated with the ASCII standard is kept together with other ASCII characters in the range U+0020..U+007E, rather than being grouped with other sets of general punctuation characters. By and large, however, the code charts are arranged so that related characters can be found near each other in the charts.

Grouping encoded characters by script or other functional categories offers the additional benefit of supporting various space-saving techniques in actual implementations, as for building tables or fonts.

For more information on writing systems, see *Section 6.1, Writing Systems*.

Planes

The Unicode codespace consists of the numeric values from 0 to $10FFFF_{16}$, but in practice it has proven convenient to think of the codespace as divided up into *planes* of characters—each plane consisting of 64K code points. The numerical sense of this is immediately obvious if one looks at the ranges of code points involved, expressed in hexadecimal. Thus, the lowest plane, the *Basic Multilingual Plane,* consists of the range 0000_{16}..$FFFF_{16}$. The next plane, the *Supplementary Multilingual Plane*, consists of the range 10000_{16}..$1FFFF_{16}$, and is also known as *Plane 1*, since the most significant hexadecimal digit for all its code positions is "1". *Plane 2*, the *Supplementary Ideographic Plane*, consists of the range 20000_{16}..$2FFFF_{16}$, and so on. Because of these numeric conventions, the Basic Multilingual Plane is also occasionally referred to as *Plane 0*.

Basic Multilingual Plane. The Basic Multilingual Plane (BMP, or Plane 0) contains all the common-use characters for all the modern scripts of the world, as well as many historical and rare characters. By far the majority of all Unicode characters for almost all textual data can be found in the BMP.

Supplementary Multilingual Plane. The Supplementary Multilingual Plane (SMP, or Plane 1) is dedicated to the encoding of lesser-used historic scripts, special-purpose invented scripts, and special notational systems, which either could not be fit into the BMP or would be of very infrequent usage. Examples of each type include Gothic, Shavian, and musical symbols, respectively. While few scripts are currently encoded in the SMP in Unicode 4.0, there are many major and minor historic scripts that do not yet have their characters encoded in the Unicode Standard, and many of those will eventually be allocated in the SMP.

Supplementary Ideographic Plane. The Supplementary Ideographic Plane (SIP, or Plane 2) is the spillover allocation area for those CJK characters that could not be fit in the blocks set aside for more common CJK characters in the BMP. While there are a small number of common-use CJK characters in the SIP (for example, for Cantonese usage), the vast majority of Plane 2 characters are extremely rare or of historical interest only.

Supplementary Special-purpose Plane. The Supplementary Special-purpose Plane (SSP, or Plane 14) is the spillover allocation area for format control characters that do not fit into the small allocation areas for format control characters in the BMP.

Allocation Areas and Character Blocks

The Unicode Standard does not have any normatively defined concept of *areas* or *zones* for the BMP (or other planes), but it is often handy to refer to the allocation areas of the BMP by the general types of the characters they include. These areas are only a rough organizational device and do not restrict the types of characters that may end up being allocated in them. The description and ranges of areas may change from version to version of the standard as more new scripts, symbols, and other characters are encoded in previously reserved ranges.

The various allocation areas are, in turn, divided up into character *blocks*, which *are* normatively defined, and which are used to structure the actual charts in *Chapter 16, Code Charts*. For a complete listing of the normative character blocks in the Unicode Standard, see Blocks.txt in the Unicode Character Database.

The normative status of character blocks should not, however, be taken as indicating that they define significant sets of characters. For the most part, the character blocks serve *only* as ranges to divide up the code charts and do not necessarily imply anything else about the types of characters found in the block. Block identity cannot be taken as a reliable guide to the source, use, or properties of characters, for example, and cannot be reliably used alone to process characters. In particular:

- Blocks are simply ranges, and many contain reserved code points.
- Characters used in a single writing system may be found in several different blocks. For example, characters used for letters for Latin-based writing systems are found in at least nine different blocks: Basic Latin, Latin-1 Supplement, Latin Extended-A, Latin Extended-B, IPA Extensions, Phonetic Extensions, Latin Extended Additional, Spacing Modifier Letters, and Combining Diacritical Marks.
- Characters in a block may be used with different writing systems. For example, the *danda* character is encoded in the Devanagari block, but is used with numerous other scripts; Arabic combining marks in the Arabic block are used with the Syriac script; and so on.
- Block definitions are not at all exclusive. For instance, there are many mathematical operator characters which are not encoded in the Mathematical Operators block—and which are not even in any block containing “Mathematical” in its name; there are many currency symbols not located in the Currency Symbols block, and so on.

For reliable specification of the properties of characters, one should instead turn to the detailed, character-by-character property assignments available in the Unicode Character Database. See also *Chapter 4, Character Properties*. For further discussion of the relationship between Unicode character blocks and significant property assignments and sets of characters, see Unicode Standard Annex #24, “Script Names,” and Unicode Technical Report #18, “Unicode Regular Expression Guidelines.”

Details of Allocation

Figure 2-13 gives an overall picture of the allocation areas of the Unicode Standard, with an emphasis on the identities of the planes.

Plane 2 consists primarily of one big area, starting from the first code point in the plane, dedicated to more unified CJK character encoding. Then there is a much smaller area, toward the end of the plane, dedicated to additional CJK compatibility ideographic characters—which are basically just duplicated character encodings required for round-trip conversion to various existing legacy East Asian character sets. The CJK compatibility ideographic characters in Plane 2 are currently all dedicated to round-trip conversion for the CNS standard and are intended to supplement the CJK compatibility ideographic characters in the BMP, a smaller number of characters dedicated to round-trip conversion for various Korean, Chinese, and Japanese standards.

Plane 14 contains a small area set aside for language tag characters, and another small area containing supplementary variation selection characters.

Figure 2-13 also shows that Plane 15 and Plane 16 are allocated, in their entirety, for private use. Those two planes contain a total of 131,068 characters, to supplement the 6,400 private-use characters located in the BMP.

Figure 2-13. Unicode Allocation

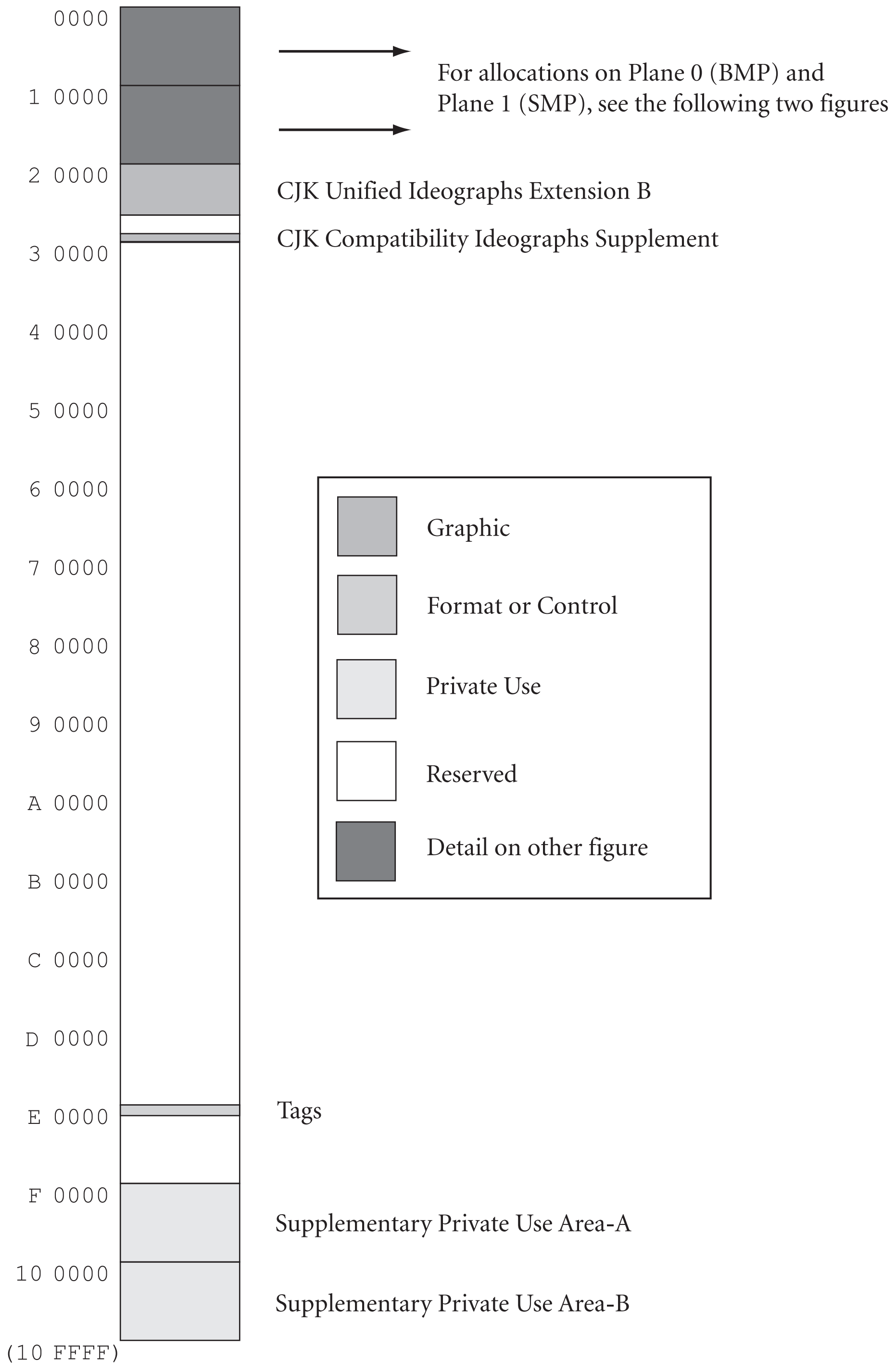
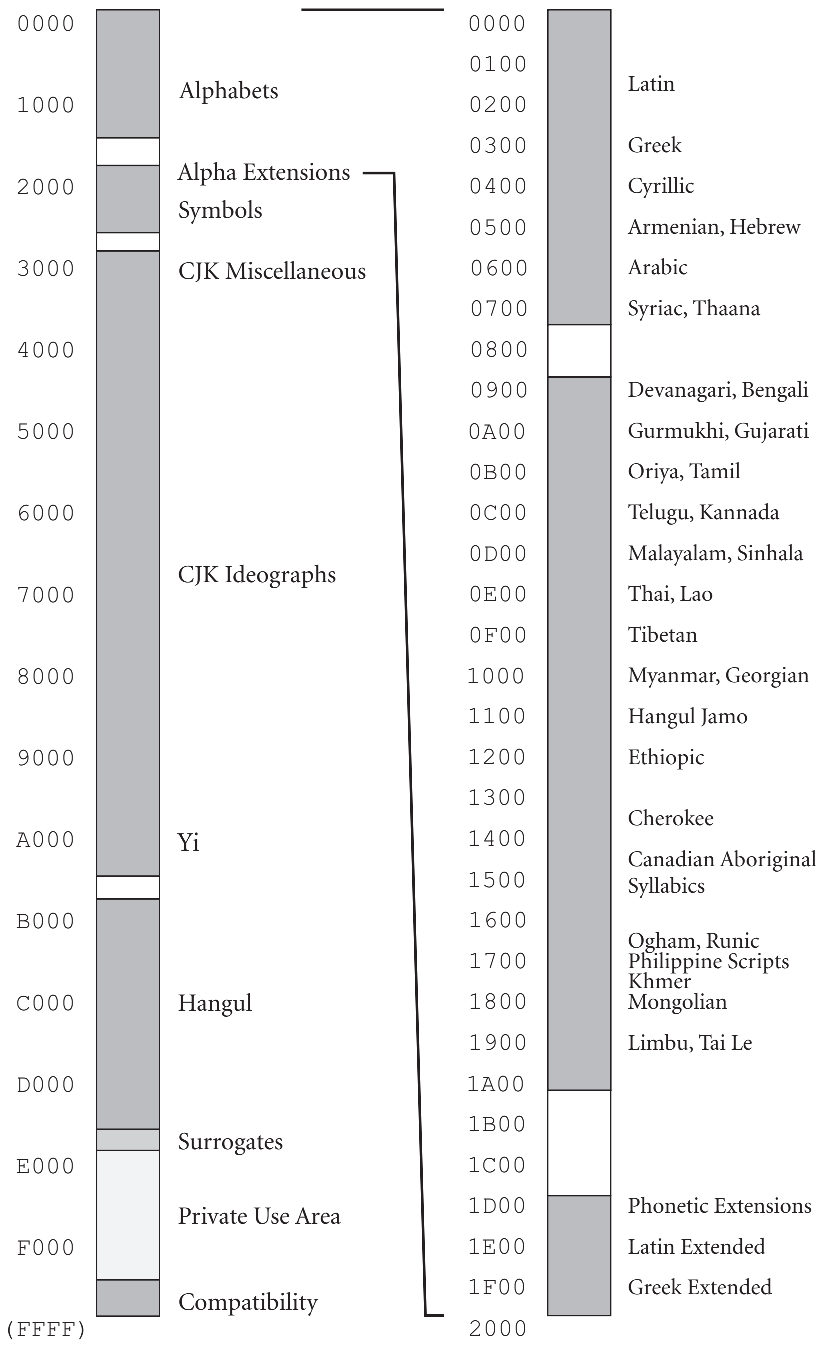

0000
1 0000
2 0000
3 0000
4 0000
5 0000
6 0000
7 0000
8 0000
9 0000
A 0000
B 0000
C 0000
D 0000
E 0000
F 0000
10 0000
(10 FFFF)
For allocations on Plane 0 (BMP) and
Plane 1 (SMP), see the following two figures
CJK Unified Ideographs Extension B
CJK Compatibility Ideographs Supplement
Graphic
Format or Control
Private Use
Reserved
Detail on other figure
Tags
Supplementary Private Use Area-A
Supplementary Private Use Area-B

Figure 2-14 shows the BMP in an expanded format to illustrate the allocation substructure of that plane in more detail.

Figure 2-14. Allocation on the BMP



The first allocation area in the BMP is the General Scripts Area. It contains a large number of modern-use scripts of the world, including Latin, Greek, Cyrillic, Arabic, and so on. This area is shown in expanded form in *Figure 2-14*. The order of the various scripts can serve as a guide to the relative positions where these scripts are found in the code charts. Most of the characters encoded in this area are graphic characters, but all 65 C0 and C1 control codes are also located here because the first two character blocks in the Unicode Standard are organized for exact compatibility with the ASCII and ISO/IEC 8859-1 standards.

A Symbols Area follows the General Scripts Area. It contains all kinds of symbols, including many characters for use in mathematical notation. It also contains symbols for punctuation as well as most of the important format control characters.

Next is the CJK Miscellaneous Area. It contains some East Asian scripts, such as Hiragana and Katakana for Japanese, punctuation typically used with East Asian scripts, lists of CJK radical symbols, and a large number of East Asian compatibility characters.

Immediately following the CJK Miscellaneous Area is the CJKV Ideographs Area. It contains all the unified Han ideographs in the BMP. It is subdivided into a block for the Unified Repertoire and Ordering (the initial block of 20,902 unified Han ideographs) and another block containing Extension A (an additional 6,582 unified Han ideographs).

The Asian Scripts Area follows the CJKV Ideographs Area. It currently contains only the Yi script and 11,172 Hangul syllables for Korean.

The Surrogates Area contains *only* surrogate code points and *no* encoded characters. See *Section 15.5, Surrogates Area*, for more details.

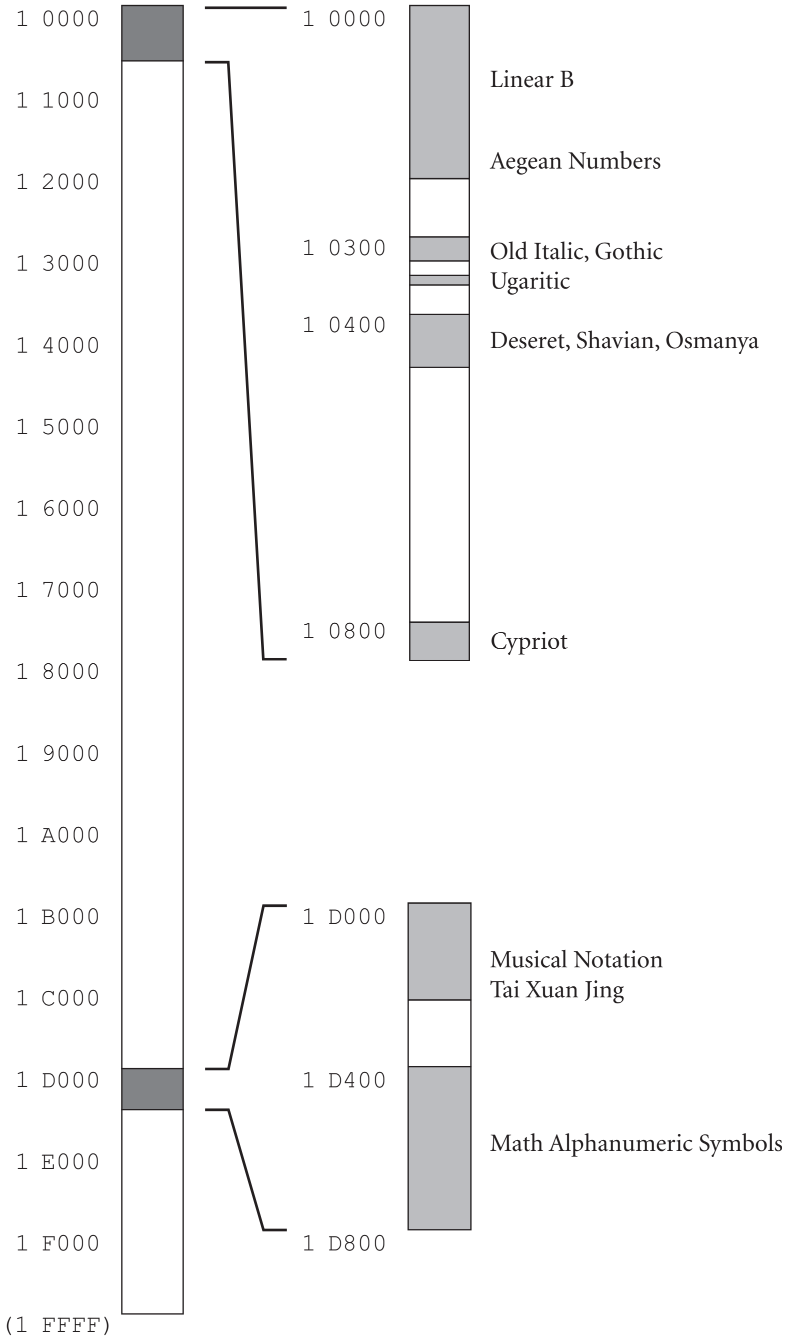
The Private Use Area in the BMP contains 6,400 private-use characters.

Finally, at the very end of the BMP, there is the Compatibility and Specials Area. It contains many compatibility characters from widely used corporate and national standards that have other representations in the Unicode Standard. For example, it contains Arabic presentation forms, whereas the basic characters for the Arabic script are located in the General Scripts Area. The Compatibility and Specials Area also contains a few important format control characters and other special characters. See *Section 15.9, Specials*, for more details.

Note that the allocation order of various scripts and other groups of characters reflects the historical evolution of the Unicode Standard. While there is a certain geographic sense to the ordering of the allocation areas for the scripts, this is only a very loose correlation. The empty spaces will be filled with future script encodings on a space-available basis. The relevant character encoding committees make use of rationally organized roadmap charts to help them decide where to encode new scripts within the available space, but until the characters for a script are actually standardized, there are no absolute guarantees where future allocations will occur. In general, implementations should not make assumptions about where future scripts may be encoded, based on the identity of neighboring blocks of characters already encoded.

Figure 2-15 shows Plane 1 in expanded format to illustrate the allocation substructure of that plane in more detail.

Figure 2-15. Allocation on Plane 1



Plane 1 currently has only two allocation areas. There is a General Scripts Area at the beginning of the plane, containing various small historic scripts. Then there is a Notational Systems Area, which currently contains sets of musical symbols, alphanumeric symbols for mathematics, and a system of divination symbols similar to those used for the *Yijing*.

Assignment of Code Points

Code points in the Unicode Standard are assigned using the following guidelines:

- Where there is a single accepted standard for a script, the Unicode Standard generally follows it for the relative order of characters within that script.
- The first 256 codes follow precisely the arrangement of ISO/IEC 8859-1 (Latin 1), of which 7-bit ASCII (ISO/IEC 646 IRV) accounts for the first 128 code positions.
- Characters with common characteristics are located together contiguously. For example, the primary Arabic character block was modeled after ISO/IEC 8859-6. The Arabic script characters used in Persian, Urdu, and other languages, but not included in ISO/IEC 8859-6, are allocated after the primary Arabic character block. Right-to-left scripts are grouped together.
- To the extent possible, scripts are allocated so as not to cross 128-code-point boundaries (that is, they fit in ranges nn00..nn7F or nn80..nnFF). For supplementary characters, an additional constraint not to cross 1,024-code-point boundaries is also applied (that is, scripts fit in ranges nn000..nn3FF, nn400..nn7FF, nn800..nnBFF, or nnC00..nnFFF). The reason for such constraints is to enable better optimizations for tasks such as building tables for access to character properties.
- Codes that represent letters, punctuation, symbols, and diacritics that are generally shared by multiple languages or scripts are grouped together in several locations.
- The Unicode Standard does not correlate character code allocation with language-dependent collation or case. For more information on collation order, see Unicode Technical Standard #10, “Unicode Collation Algorithm.”
- Unified CJK ideographs are laid out in three sections, each of which is arranged according to the Han ideograph arrangement defined in *Section 11.1, Han*. This ordering is roughly based on a radical-stroke count order.

2.9 Writing Direction

Individual writing systems have different conventions for arranging characters into lines on a page or screen. Such conventions are referred to as a script’s *directionality*. For example, in the Latin script, characters run horizontally from left to right to form lines, and lines run from top to bottom.

In Semitic scripts such as Hebrew and Arabic, characters are arranged from right to left into lines, although digits run the other way, making the scripts inherently bidirectional. Left-to-right and right-to-left scripts are frequently used together. In such a case, arranging characters into lines becomes more complex. The Unicode Standard defines an algorithm to determine the layout of a line. See Unicode Standard Annex #9, “The Bidirectional Algorithm,” for more information.

East Asian scripts are frequently written in vertical lines that run from top to bottom. Lines are arranged from right to left, except for Mongolian, for which lines proceed from left to right. Such scripts may also be written horizontally, left to right. Most characters have the same shape and orientation when displayed horizontally or vertically, but many punctuation characters change their shape when displayed vertically. In a vertical context, letters and words from other scripts are generally rotated through 90-degree angles so that they, too, read from top to bottom. That is, letters from left-to-right scripts will be rotated clockwise and letters from right-to-left scripts will be rotated counterclockwise.

In contrast to the bidirectional case, the choice to lay out text either vertically or horizontally is treated as a formatting style. Therefore, the Unicode Standard does not provide directionality controls to specify that choice.

Other script directionalities are found in historical writing systems. For example, some ancient Numidian texts are written bottom to top, and Egyptian hieroglyphics can be written with varying directions for individual lines.

Early Greek used a system called *boustrophedon* (literally, "ox-turning"). In boustrophedon writing, characters are arranged into horizontal lines, but the individual lines alternate between running right to left and running left to right, the way an ox goes back and forth when plowing a field. The letter images are mirrored in accordance with the direction of each individual line.

The historical directionalities are of interest almost exclusively to scholars intent on reproducing the exact visual content of ancient texts. The Unicode Standard does not provide direct support for them. Fixed texts can, however, be written in boustrophedon or in other directional conventions by using hard line breaks and directionality overrides.

2.10 Combining Characters

Combining Characters. 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 property lists as "combining" or "nonspacing" characters. When rendered, the glyphs that depict these characters are intended to be positioned relative to the glyph depicting the preceding base character in some combination. The Unicode Standard distinguishes two types of combining characters: spacing and nonspacing. Nonspacing combining characters do not occupy a spacing position by themselves. In rendering, the combination of a base character and a nonspacing character may have a different advance width than the base character by itself. For example, an " î " may be slightly wider than a plain "i". The spacing or nonspacing properties of a combining character are defined in the Unicode Character Database.

Diacritics. Diacritics are the principal class of nonspacing combining characters used with European alphabets. In the Unicode Standard, the term "diacritic" is defined very broadly to include accents as well as other nonspacing marks.

All diacritics can be applied to any base character and are available for use with any script. A separate block is provided for symbol diacritics, generally intended to be used with symbol base characters. Other blocks contain additional combining characters for particular scripts with which they are primarily used. As with other characters, the allocation of a combining character 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.*

Other Combining Characters. Some scripts, such as Hebrew, Arabic, and the scripts of India and Southeast Asia, have spacing or nonspacing combining characters. Many of these combining marks encode vowel letters; as such, they are not generally referred to as "diacritics."

Sequence of Base Characters and Diacritics

In the Unicode Standard, all combining characters are to be used in sequence following the base characters to which they apply. The sequence of Unicode characters U+0061 “a” LATIN SMALL LETTER A + U+0308 “◌̈”COMBINING DIAERESIS + U+0075 “u” LATIN SMALL LETTER U unambiguously encodes “äu” not “aü”.

The ordering convention used by the Unicode Standard—placing combining marks after the base character to which they apply—is consistent with the logical order of combining characters in Semitic and Indic scripts, the great majority of which (logically or phonetically) follow the base characters with respect to which they are positioned. This convention conforms to the way modern font technology handles the rendering of nonspacing graphical forms (glyphs) so that mapping from character memory representation order to font rendering order is simplified. It is different from the convention used in the bibliographic standard ISO 5426.

A sequence of a base character plus one or more combining characters generally has the same properties as the base character. For example, “A” followed by “ˆ ” has the same properties as “Â”. In some contexts, enclosing diacritics confer a symbol property to the characters they enclose. This idea is discussed more fully in *Section 3.11, Canonical Ordering Behavior*, but see also Unicode Standard Annex #9, “The Bidirectional Algorithm.”

In the charts for Indic scripts, some vowels are depicted to the left of dotted circles (see *Figure 2-16*). This special case must be carefully distinguished from that of general combining diacritical mark 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 follows the consonant letter. The coding of these vowels in pronunciation order and not in visual order is consistent with the ISCII standard.

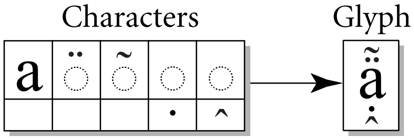
Figure 2-16. Indic Vowel Signs

फ + ि◌ → फि

Multiple Combining Characters

In some instances, more than one diacritical mark is applied to a single base character (see *Figure 2-17*). The Unicode Standard does not restrict the number of combining characters that may follow a base character. The following discussion summarizes the default treatment of multiple combining characters. (For the formal algorithm, see *Chapter 3, Conformance.*)

Figure 2-17. Stacking Sequences



If the combining characters can interact typographically—for example, a U+0304 COMBINING MACRON and a U+0308 COMBINING DIAERESIS—then the order of graphic display is determined by the order of coded characters (see *Figure 2-18*). The diacritics or other combining characters are positioned from the base character’s glyph outward. Combining characters 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 combining characters placed below a base character, the situation is reversed, with the combining characters starting from the base character and stacking downward.

When combining characters do not interact typographically, the relative ordering of contiguous combining marks cannot result in any visual distinction and thus is insignificant.

Figure 2-18. Interaction of Combining Characters

Glyph	Equivalent Sequences
ã	LATIN SMALL LETTER A WITH TILDE LATIN SMALL LETTER A + COMBINING TILDE
ȧ	LATIN SMALL LETTER A + COMBINING DOT ABOVE
ạ̃	LATIN SMALL LETTER A WITH TILDE + COMBINING DOT BELOW LATIN SMALL LETTER A + COMBINING TILDE + COMBINING DOT BELOW LATIN SMALL LETTER A + COMBINING DOT BELOW + COMBINING TILDE
ạ̇	LATIN SMALL LETTER A + COMBINING DOT BELOW + COMBINING DOT ABOVE LATIN SMALL LETTER A + COMBINING DOT ABOVE + COMBINING DOT BELOW
ấ	LATIN SMALL LETTER A WITH CIRCUMFLEX AND ACUTE LATIN SMALL LETTER A WITH CIRCUMFLEX + COMBINING ACUTE LATIN SMALL LETTER A + COMBINING CIRCUMFLEX + COMBINING ACUTE
á̂	LATIN SMALL LETTER A WITH ACUTE + COMBINING CIRCUMFLEX LATIN SMALL LETTER A + COMBINING ACUTE + COMBINING CIRCUMFLEX

An example of multiple combining 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 that produces this graphic display is *base consonant character* + *vowel character* + *tone mark character*.

Some specific uses of combining characters override the default stacking behavior by being positioned horizontally rather than stacking or by ligature with an adjacent nonspacing mark (see *Figure 2-19*). When positioned horizontally, the order of codes is reflected by positioning in the predominant direction of the script with which the codes are used. For example, in a left-to-right script, horizontal accents would be coded left to right. In *Figure 2-19*, the top example is correct and the bottom example is incorrect.

Such override behavior is associated with specific scripts or alphabets. For example, when used with the Greek script, the "breathing marks" U+0313 combining comma above (*psili*) and U+0314 combining reversed comma above (*dasia*) require that, when used together with a following acute or grave accent, they be rendered side-by-side rather than the accent marks being stacked above the breathing marks. The order of codes here is *base character code* + *breathing mark code* + *accent mark code*. This example demonstrates the script-dependent or writing system-dependent nature of rendering combining diacritical marks.

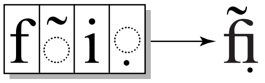
Figure 2-19. Nondefault Stacking

ἄ	GREEK SMALL LETTER ALPHA + COMBINING COMMA ABOVE (psili) + COMBINING ACUTE ACCENT (oxia)	This is correct
ά̓	GREEK SMALL LETTER ALPHA + COMBINING ACUTE ACCENT (oxia) + COMBINING COMMA ABOVE (psili)	This is incorrect

Ligated Multiple Base Characters

When the glyphs representing two base characters merge to form a ligature, then the combining characters must be rendered correctly in relation to the ligated glyph (see *Figure 2-20*). Internally, the software must distinguish between the nonspacing marks that apply to positions relative to the first part of the ligature glyph and those that apply to the second. (For a discussion of general methods of positioning nonspacing marks, see *Section 5.12, Strategies for Handling Nonspacing Marks*.)

Figure 2-20. Ligated Multiple Base Characters



For more information, see the subsection on “Application of Combining Marks,” in *Section 3.11, Canonical Ordering Behavior*.

Ligated base characters with multiple combining marks do not commonly occur in most scripts. However, in some scripts, such as Arabic, this situation occurs quite often when vowel marks are used. It arises because of the large number of ligatures in Arabic, where each element of a ligature is a consonant, which in turn can have a vowel mark attached to it. Ligatures can even occur with three or more characters merging; vowel marks may be attached to each part.

Spacing Clones of European Diacritical Marks

By convention, diacritical marks used by the Unicode Standard may be exhibited in (apparent) isolation by applying them to U+0020 SPACE or to U+00A0 NO-BREAK SPACE. This tactic might be employed, 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 set standards. These related characters are cross-referenced in the names list in *Chapter 16, Code Charts*.

“Characters” and Grapheme Clusters

End users have various concepts about what constitutes a letter or “character” in the writing system for their language or languages. Such concepts are often important for processes such as collation, for the definition of regular expressions, and for counting “character” positions within text. In instances such as these, what the user thinks of as a character may affect how the collation or regular expression will be defined or how the “characters” will be counted. Words and other higher-level text elements generally do not split within elements that a user thinks of as a character, even when the Unicode representation of them may consist of a sequence of encoded characters. The precise scope of these end-user “characters” depends on the particular written language and the orthography it uses. In addition to the many instances of accented letters, they may extend to digraphs such as Slovak “ch”, trigraphs or longer combinations, and sequences using spacing letter modifiers, such as “k^w”.

The variety of these end-user perceived characters is quite great—particularly for digraphs, ligatures, or syllabic units. Furthermore, it depends on the particular language and writing system that may be involved. Despite this variety, however, there is a core concept of “characters that should be kept together” that can be defined for the Unicode Standard in a language-independent way. This core concept is known as a *grapheme cluster*, and it consists of any combining character sequence that contains only *nonspacing* combining marks, or any sequence of characters that constitutes a Hangul syllable (possibly followed by one or more nonspacing marks). An implementation operating on such a cluster would almost never want to break between its elements for rendering, editing, or other such text process; the grapheme cluster is treated as a single unit. Unicode Standard Annex #29, “Text Boundaries,” provides a complete formal definition of a grapheme cluster and discusses its application in the context of editing and other text processes. Implementations also may tailor the definition of a grapheme cluster, so that under limited circumstances, particular to one written language or another, the grapheme cluster may more closely pertain to what end users think of as “characters” for that language.

2.11 Special Characters and Noncharacters

The Unicode Standard includes a small number of important characters with special behavior; some of them are introduced in this section. It is important that implementations treat these characters properly. For a list of these and similar characters, see *Section 4.11, Characters with Unusual Properties*; for more information about such characters, see *Section 15.1, Control Codes*; *Section 15.2, Layout Controls*; *Section 15.8, Noncharacters*; and *Section 15.9, Specials*.

Byte Order Mark (BOM)

The UTF-16 and UTF-32 encoding forms of Unicode plain text are sensitive to the byte ordering that is used when serializing text into a sequence of bytes, such as when writing to a file or transferring across a network. Some processors place the least significant byte in the initial position; others place the most significant byte in the initial position. Ideally, all implementations of the Unicode Standard would follow only one set of byte order rules, but this scheme would force one class of processors to swap the byte order on reading and writing plain text files, even when the file never leaves the system on which it was created.

To have an efficient way to indicate which byte order is used in a text, the Unicode Standard contains two code points, U+FEFF ZERO WIDTH NO-BREAK SPACE (*byte order mark*) and U+FFFE (not a character code), which are the byte-ordered mirror images of one another. When a byte order mark is received with the opposite byte order, it will be recognized as a noncharacter and can therefore be used to detect the intended byte order of the text. The *byte order mark* is not a control character that selects the byte order of the text; rather, its function is to allow recipients to determine which byte ordering is used in a file.

Unicode Signature. An initial BOM may also serve as an implicit marker to identify a file as containing Unicode text. For UTF-16, the sequence FE_{16} FF_{16} (or its byte-reversed counterpart, FF_{16} FE_{16}) is exceedingly rare at the outset of text files that use other character encodings. The corresponding UTF-8 BOM sequence, EF_{16} BB_{16} BF_{16}, is also exceedingly rare. In either case, it is therefore unlikely to be confused with real text data. The same is true for both single-byte and multibyte encodings.

Data streams (or files) that begin with U+FEFF *byte order mark* are likely to contain Unicode characters. It is recommended that applications sending or receiving untyped data streams of coded characters use this signature. If other signaling methods are used, signatures should not be employed.

Conformance to the Unicode Standard does not requires the use of the BOM as such a signature. See *Section 15.9, Specials*, for more information on the *byte order mark* and its use as an encoding signature.

Special Noncharacter Code Points

The Unicode Standard contains a number of code points that are intentionally *not* used to represent assigned characters. These code points are known as *noncharacters*. They are permanently reserved for internal use and should never be used for open interchange of Unicode text. For more information on noncharacters, see *Section 15.8, Noncharacters.*

Layout and Format Control Characters

The Unicode Standard defines several characters that are used to control joining behavior, bidirectional ordering control, and alternative formats for display. These characters are explicitly defined as not affecting line-breaking behavior. Unlike space characters or other delimiters, they do not serve to indicate word, line, or other unit boundaries. Their specific use in layout and formatting is described in *Section 15.2, Layout Controls.*

The Replacement Character

U+FFFD REPLACEMENT CHARACTER is the general substitute character in the Unicode Standard. It can be substituted for any "unknown" character in another encoding that cannot be mapped in terms of known Unicode characters (see *Section 5.3, Unknown and Missing Characters*, and *Section 15.9, Specials*).

Control Codes

In addition to the special characters defined in the Unicode Standard for a number of purposes, the standard incorporates the legacy control codes for compatibility with the ISO/IEC 2022 framework, ASCII, and the various protocols that make use of control codes. Rather than simply being defined as byte values, however, the legacy control codes are assigned to Unicode code points: U+0000..U+001F, U+007F..U+009F. Those code points for control codes must be represented consistently with the various Unicode encoding forms when they are used with other Unicode characters. For more information on control codes, see *Section 15.1, Control Codes.*

2.12 Conforming to the Unicode Standard

Chapter 3, Conformance, specifies the set of unambiguous criteria to which a Unicode-conformant implementation must adhere so that it can interoperate with other conformant implementations. This section gives examples of conformance and nonconformance to complement the formal statement of conformance.

An implementation that conforms to the Unicode Standard has the following characteristics:

- It treats characters according to the specified Unicode encoding form.

  <20 20> is interpreted as U+2020 '†' DAGGER in the UTF-16 encoding form.

  <20 20> is interpreted as the sequence <U+0020, U+0020>, two spaces, in the UTF-8 encoding form.

- It interprets characters according to the identities, properties, and rules defined for them in this standard.

  U+2423 is '␣' OPEN BOX, *not* 'ぃ' *hiragana small i* (which is the meaning of the bytes 2423_{16} in JIS).

  U+00F4 'ô' is equivalent to U+006F 'o' followed by U+0302 '◌̂', but *not equivalent to* U+0302 followed by U+006F.

  U+05D0 'א' followed by U+05D1 'ב' looks like 'אב', *not* 'בא' when displayed.

  When an implementation supports Arabic or Hebrew characters and displays those characters, they must be ordered according to the bidirectional algorithm described in Unicode Standard Annex #9, "The Bidirectional Algorithm."

  When an implementation supports Arabic, Devanagari, Tamil, or other shaping characters and displays those characters, at a minimum the characters are shaped according to the appropriate character block descriptions given in *Section 8.2, Arabic*; *Section 9.1, Devanagari*; or *Section 9.6, Tamil.* (More sophisticated shaping can be used if available.)

- It does not use unassigned codes.

  U+2073 is unassigned and not usable for '^{3}' (*superscript 3*) or any other character.

- It does not corrupt unknown characters.

  U+2029 is PARAGRAPH SEPARATOR and should not be dropped by applications that do not yet support it.

  U+03A1 "P" GREEK CAPITAL LETTER RHO should not be changed to U+00A1 (first byte dropped), U+0050 (mapped to Latin letter *P*), U+A103 (bytes reversed), or anything other than U+03A1.

However, it is acceptable for a conforming implementation:

- To support only a subset of the Unicode characters.

  An application might not provide mathematical symbols or the Thai script, for example.

- To transform data knowingly.

  Uppercase conversion: 'a' transformed to 'A'

  Romaji to kana: 'kyo' transformed to きょ

  U+247D '(10)' decomposed to 0028 0031 0030 0029

- To build higher-level protocols on the character set.

  Compression of characters

  Use of rich text file formats

- To define private-use characters.

  Examples of characters that might be defined for private use include additional ideographic characters (*gaiji*) or existing corporate logo characters.

- To not support the bidirectional algorithm or character shaping in implementations that do not support complex scripts, such as Arabic and Devanagari.

- To not support the bidirectional algorithm or character shaping in implementations that do not display characters, such as on servers or in programs that simply parse or transcode text, such as an XML parser.

Code conversion between other character encodings and the Unicode Standard will be considered conformant if the conversion is accurate in both directions.

Supported Subsets

The Unicode Standard does not require that an application be capable of interpreting and rendering all Unicode characters so as to be conformant. Many systems will have fonts only for some scripts, but not for others; sorting and other text-processing rules may be implemented only for a limited set of languages. As a result, an implementation is able to interpret a subset of characters.

The Unicode Standard provides no formalized method for identifying an implemented subset. Furthermore, such a subset is typically different for different aspects of an implementation. For example, an application may be able to read, write, and store any Unicode character, and to sort one subset according to the rules of one or more languages (and the rest arbitrarily), but have access only to fonts for a single script. The same implementation may be able to render additional scripts as soon as additional fonts are installed in its environment. Therefore, the subset of interpretable characters is typically not a static concept.

Conformance to the Unicode Standard implies that whenever text purports to be unmodified, uninterpreted code points must not be removed or altered. (See also *Section 3.2, Conformance Requirements.*)

2.13 Related Publications

In addition to the Unicode Standard, the Unicode Consortium publishes Unicode Technical Standards and Unicode Technical Reports. These are published as electronic documents only.

A Unicode Technical Standard (UTS) is a separate specification with its own conformance requirements. Any UTS may include a requirement for an implementation of the UTS to also conform to a specific, base level of the Unicode Standard, but conformance to the Unicode Standard as such does not require conformance to any UTS.

A Unicode Technical Report (UTR) contains informative material. Unicode Technical Reports do not contain conformance requirements of their own, nor does conformance to the Unicode Standard require conformance to any specifications contained in any of the UTRs. Other specifications, however, are free to cite the material in UTRs and to make any level of conformance requirements within their own context.

There is a third kind of electronic document called a Unicode Standard Annex (UAX), which is defined in *Section 3.2, Conformance Requirements*. Unicode Standard Annexes differ from UTRs and UTSs in that they form an integral part of the Unicode Standard. For a summary overview of important Unicode Technical Standards, Unicode Technical Reports, and Unicode Standard Annexes, see *Appendix B, Abstracts of Unicode Technical Reports*.