Chapter 2

General Structure

This chapter discusses the fundamental principles governing the design of the Unicode Standard and presents an overview of its main features. It includes discussion of text processes, unification principles, allocation of codespace, character properties, writing direction, and a description of combining marks and how they are employed in Unicode character encoding. This chapter also gives general requirements for creating a text-processing system that conforms to the Unicode Standard. Formal requirements for conformance appear in *Chapter 3, Conformance*. Character properties, both normative and informative, are given in *Chapter 4, Character Properties*. A set of guidelines for implementers is provided in *Chapter 5, Implementation Guidelines*.

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
- Modifying 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, Code Values, and Text Processes

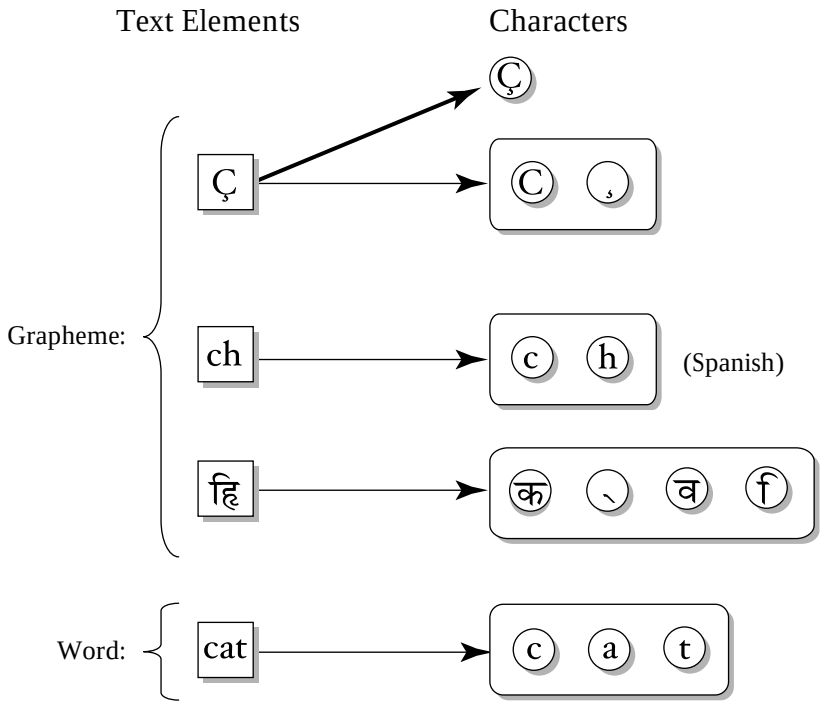
One of the more profound challenges in designing a worldwide character encoding stems from the fact that, for each text process, written languages differ in what is considered a fundamental unit of text, or a *text element*.

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 objects “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.

A character encoding standard provides the fundamental units of encoding (that is, the abstract characters), which must exist in a unique relationship to the assigned numerical *code values*. These code values are the smallest addressable units of stored text.

An important class of text elements is called a *grapheme*, which typically corresponds to what a user thinks of as a “character.” *Figure 2-1* illustrates the relationship between abstract characters and graphemes.

Figure 2-1. Text Elements and Characters



The design of the character encoding must provide precisely the set of code values that allows programmers to design applications capable of implementing a variety of text processes in the desired languages. These code values 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 English fail for other writing systems. Unlike in English, characters in 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*, in *Chapter 8, Middle Eastern Scripts*, and *Section 9.1, Devanagari*, in *Chapter 9, South and Southeast Asian Scripts*, 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 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 *Section 5.17, Sorting and Searching*, for implementation guidelines.)

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
Sixteen-bit character codes	Unicode character codes have a width of 16 bits.
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	The Unicode Standard encodes 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 sequence	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.

Sixteen-Bit Character Codes

Plain Unicode text consists of sequences of 16-bit Unicode character codes. Sequences of 16-bit codes are easy to parse and allow for efficient sorting, searching, display, and editing of text. From the full range of 65,536 code values, 63,486 are available to represent characters with single 16-bit code values, and 2,048 code values are available to represent an additional 1,048,544 characters through paired 16-bit code values. These paired code values, or surrogates, will allow implementations access to additional characters in the future. *None of these surrogate pairs has been assigned in this version of the standard.*

The unrestricted range of 256 byte values is used for either of the two bytes that compose a Unicode 16-bit code value. An 8-bit-oriented process that expects certain ranges of byte values to be reserved may fail if unexpectedly presented with Unicode's 16-bit character codes. For compatibility with existing environments, a lossless transformation for converting 16-bit Unicode values into a form appropriate for 8-bit environments has been defined; it is called UTF-8.

UTF-8 (Unicode Transformation Format-8) is the standard method for transforming Unicode values into a sequence of 8-bit codes. It is intended to be used where 8-bit codes are needed and/or when the ASCII range of values need to be preserved—for example, when transmitting data through byte-oriented protocols or when using byte-oriented APIs.

For further information on transformation formats of Unicode, see *Section 2.3, Encoding Forms*, and *Appendix C.3, UCS Transformation Formats*, in this book. Also see ISO/IEC 10646 Transformation Formats. For a format to be used on EBCDIC-based systems, see Unicode Technical Report #16, "UTF-EBCDIC," on the CD-ROM or the up-to-date version on the Unicode Web site.

Efficiency

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

All encoding forms are self-synchronizing. 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 low-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.

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. The common punctuation characters are shared.

Formatting characters are given specific and unambiguous function 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*, which are the smallest components of written language that have semantic value, and *glyphs*, which represent the shapes that characters can have when they are rendered or displayed. 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*.

Unicode characters represent primarily, but not exclusively, the letters, punctuation, and other signs that constitute natural language text and technical notation. Characters are represented by code values that reside only in a memory representation, as strings in memory, or on disk. The Unicode Standard deals only with character codes.

In contrast to characters, glyphs appear on the screen or paper as particular representations of one or more 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.

Figure 2-2. Characters Versus Glyphs

Glyphs	Unicode Characters	
A *A* **A** 𝔄 A 𝒜 A **A**	U+0041	LATIN CAPITAL LETTER A
a *a* **a** 𝔞 *a* 𝒶 a **a**	U+0061	LATIN SMALL LETTER A
fi fi	U+0066 + U+0069	LATIN SMALL LETTER F LATIN SMALL LETTER I
ه ﻫ ﻬ ﻪ	U+0647	ARABIC LETTER HEH

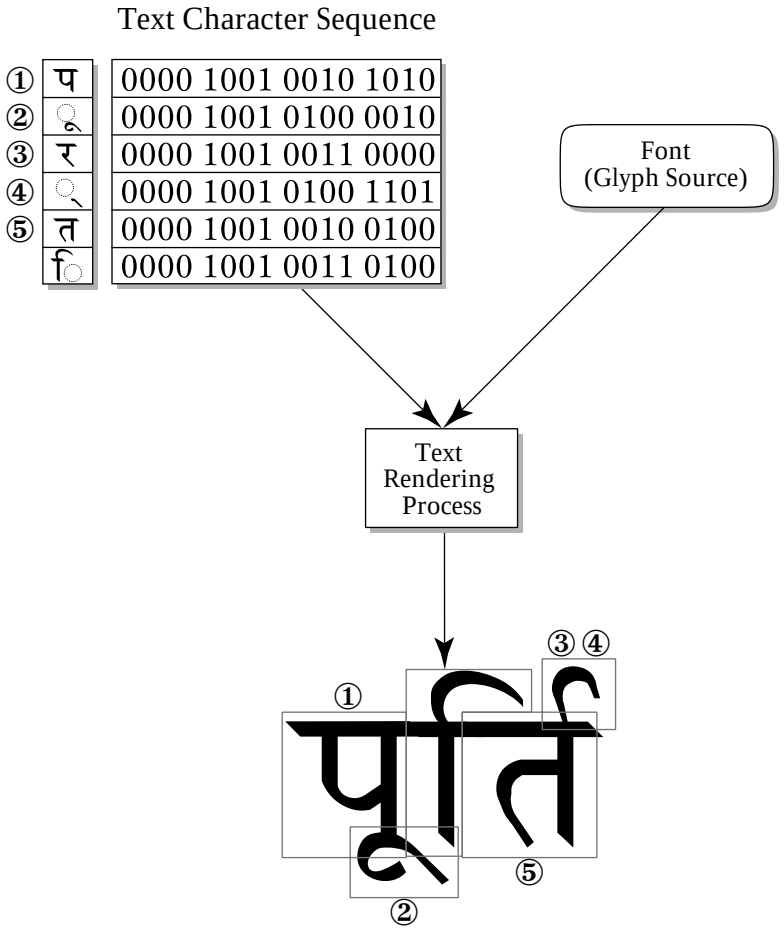
For certain scripts, such as Arabic and the various Indic scripts, the number of glyphs needed to display a given script may be significantly larger than the number of characters encoding the basic units of that script. The number of glyphs may also depend on the orthographic style supported by the font. 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 values 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*.

Figure 2-3. Unicode Character Code to Rendered Glyphs



For all scripts, an archetypical relation exists between character code sequences and resulting glyphic appearance. For the Latin script, this relationship is 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 solely for the purpose of ensuring that the same text content is always stored with the same, and therefore interchangeable, sequence of character codes.

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. See *Section 5.15, Locating Text Element Boundaries*, *Section 5.16, Identifiers*, and *Section 5.17, Sorting and Searching*, for suggested implementations. 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. For an exception, see *Section 4.1, Case—Normative*.

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, *fancy 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 fancy text.

Many kinds of data structures can be built into fancy text. For example, in fancy text containing ideographs an application may store the phonetic readings of ideographs somewhere in the fancy text structure.

The simplicity of plain text gives it a natural role as a major structural element of fancy text. SGML, HTML, XML, or TEX are examples of fancy text fully represented as plain text streams, interspersing plain text data with sequences of characters that represent the additional data structures. Many popular word processing packages rely on a buffer of plain text to represent the content and to implement links to a parallel store of formatting data.

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

- Plain text is the underlying content stream to which formatting can be applied.
- Plain text is public, standardized, and universally readable.
- Fancy text representation may be implementation-specific or proprietary.

Although some fancy text formats have been standardized or made public, the majority of fancy text designs are vehicles for particular implementations and are not necessarily readable by other implementations. Given that fancy text equals plain text plus added information, the extra information in fancy text can always be stripped away to reveal the "pure" text underneath. This operation is often employed, for example, in word processing systems that use both their own private fancy 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*.

Standards for markup language, such as XML and HTML, use plain text for the entire file. They use special conventions embedded within the plain text file such as "<p>" to distinguish the markup from the "real" content. XML, in particular, uses Unicode as a base-level encoding: "All XML processors must be able to read entities in either UTF-8 or UTF-16."—*W3C Recommendation: Extensible Markup Language (XML) 1.0, §4.3.3*. Files not using these character sets in XML must have an encoding declaration to indicate any other character set.

Because plain text represents character content, it has no inherent appearance. It requires a rendering process to make it visible. If the same plain text sequence is given to disparate rendering processes, there is no expectation that rendered text in each instance should have the same appearance. Instead, the disparate rendering processes are simply required to

make the text legible according to the intended reading. Therefore, the relationship between appearance and content of plain text may be stated as follows:

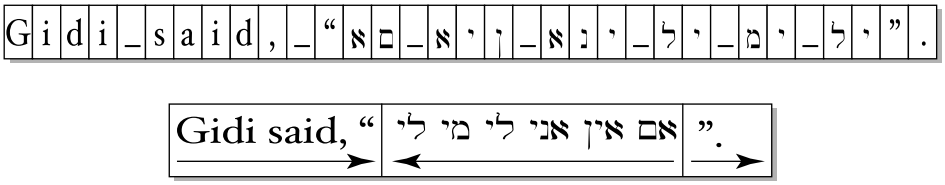
> *Plain text must contain enough information to permit the text to be rendered legibly, and nothing more.*

The Unicode Standard encodes plain text. The distinction between data encoded in the Unicode Standard 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. The 64 control code positions of ISO/IEC 6429 (commonly used with ISO/IEC 646 and ISO/IEC 8859) are retained for compatibility and may be used to implement such protocols. (See *Section 2.8, Controls and Control Sequences.*)

Logical Order

For all scripts, 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 2-4.*

Figure 2-4. Bidirectional Ordering



When the text in *Figure 2-4* 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. This inherent directionality is occasionally insufficient to render plain text legibly, however. This situation can arise when scripts of different directionality are mixed. For this reason, the Unicode Standard includes characters to specify changes in direction. *Chapter 3, Conformance,* provides rules for the correct presentation of text containing left-to-right and right-to-left scripts.

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 code string, all such characters *follow* the base character that they

modify (for example, Roman "ã" is stored as "a" followed by combining "◌̃" when not stored in a precomposed form).

Unification

The Unicode Standard avoids duplicate encoding of characters by unifying them within scripts across languages; characters that are equivalent in form 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 10.1, Han.*)

Care has been taken not to make artificial distinctions among characters. Users may become confused when they see an *Å* on the screen but their search dialog does not find it. The reason that this situation occurs is that what they see on the screen is *not* an *Å* (*A-ring*)—it is an *Å* (*Ångström*). This phenomenon is called visual ambiguity.

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.

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 grecque*, 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 10.1, Han*. Where these principles determine that two forms constitute a trivial (*wazukana*) difference, the Unicode Standard assigns a single code. Otherwise, separate codes are assigned.

Compatibility Characters. Compatibility characters are those that would not have been encoded (except for compatibility) because they are in some sense variants of characters that have already been coded. The prime examples are the glyph variants in the Compatibility Area: halfwidth characters, Arabic contextual form glyphs, Arabic ligatures, and so on.

The Compatibility Area contains a large number of compatibility characters, but the Unicode Standard also contains many compatibility characters that do not appear in the Compatibility Area. Examples include Roman numerals, such as the IV "character." It is important to be able to identify which characters are compatibility characters so that Unicode-based systems can treat them in a uniform way.

Identifying one character as a compatibility variant of another character implies that generally the first can be remapped to the other without the loss of any information other than formatting. Such remapping cannot always take place because many of the compatibility characters are in place just to allow systems to maintain one-to-one mappings to existing code sets. In such cases, a remapping would lose information that is felt to be important in the original set. Compatibility mappings are called out in *Section 14.1, Character Names List*. Because replacing a character by its compatibly equivalent character or character sequence may change the information in the text, implementation must proceed with caution. A good use of these mappings may not be in transcoding, but rather in providing the correct equivalence for searching and sorting.

Dynamic Composition

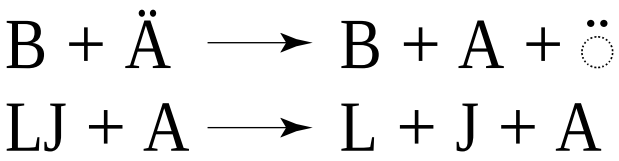
The Unicode Standard allows for the dynamic composition of accented forms and Hangul syllables. Combining characters used to create composite forms are productive. Because the process of character composition is open-ended, new forms with modifying marks may be created from a combination of base characters 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 Sequence

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 the canonically equivalent dynamically composed sequence of characters. (See also *Section 3.6, Decomposition*.)

In many cases, different sequences of Unicode characters are considered equivalent. For example, a precomposed character may be represented as a composed character sequence (see *Figure 2-5* and *Figure 2-10*).

Figure 2-5. Equivalent Sequences



In such cases, the Unicode Standard does not prescribe one particular sequence; all of the sequences in the examples are equivalent. Systems may choose to normalize Unicode text to one particular sequence, such as by normalizing composed character sequences into precomposed characters, or vice versa. Therefore, any distinctions made between nonidentical equivalent sequences by applications or users are not guaranteed to be interchangeable. (For implementation guidelines, see *Section 5.7, Normalization*.)

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 value in another standard will correspond to a single code value in the Unicode Standard. Sometimes, however, a single code value in another standard corresponds to a sequence of code values 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*.)

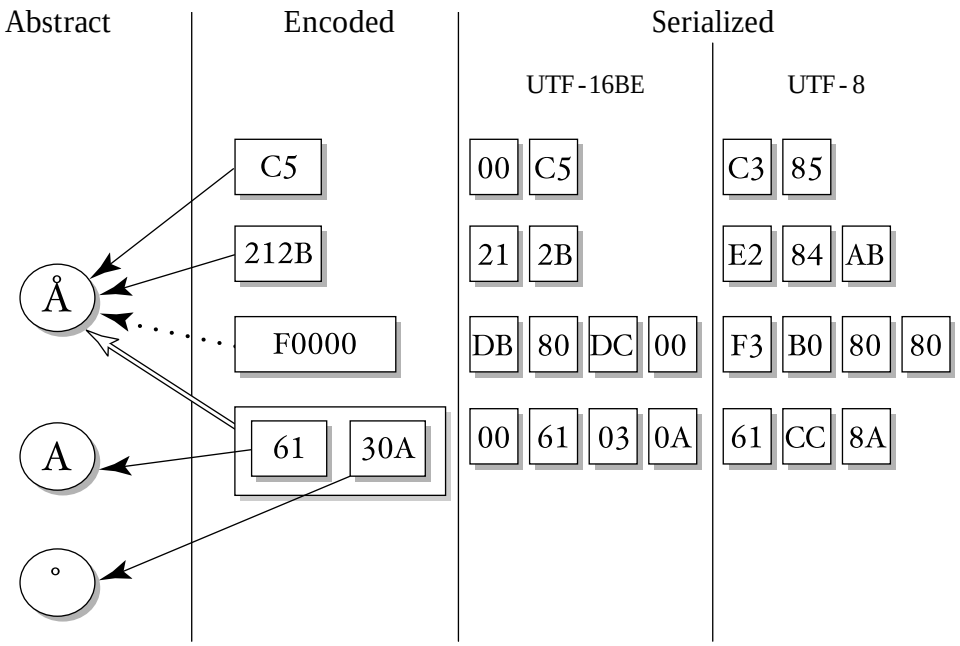
2.3 Encoding Forms

The Unicode Standard character encoding model includes a repertoire of characters, their mapping to a set of integers, and their encoding forms. *Character encoding forms* specify the representation of characters as actual data in a computer. The Unicode Standard uses two encoding forms: 16-bit and 8-bit. For a formal definition of these encoding forms, see *Section 3.8, Transformations.*

Figure 2-6 illustrates the relationship between abstract characters, encoded characters, and character encoding schemes. In *Figure 2-6*:

- The solid arrows connect encoded characters with the abstract characters that they represent and encode.
- The dotted arrow illustrates a hypothetical private-use variant of the A-ring character (the value $F0000_{16}$ is the Unicode scalar value for a surrogate pair <U+DB80,U+DC00> in the Private Use Area).
- The hollow arrow shows where an encoded character sequence represents an abstract character, but does not directly encode it.
- The serialization column shows two of the possible serializations of the encoded characters.

Figure 2-6. Character Encoding Schemes



UTF-16

The default encoding form of the Unicode Standard is 16-bit: characters are assigned a unique 16-bit value, except that characters encoded by surrogates consist of a pair of 16-bit values. The Unicode 16-bit encoding form is identical to the ISO/IEC 10646 transformation format UTF-16. In UTF-16, characters mapped up to 65535 are encoded as single 16-bit values; characters mapped above 65535 are encoded as pairs of 16-bit values (surrogates). Surrogates are defined in *Section 3.7, Surrogates*.

By using surrogates, UTF-16 provides a coded representation for more than 1 million graphic characters in a form that is compatible with 16-bit characters. This scheme permits the coexistence of a very large number of characters in systems that define characters as 16-bit entities. Surrogates were designed to be a simple and efficient extension mechanism that works well with older implementations and avoids many of the problems of multibyte encodings. See *Section 5.4, Handling Surrogate Pairs*, for more information about surrogate characters.

UTF-8

To meet the requirements of byte-oriented, ASCII-based systems, a second encoding form is specified by the Unicode Standard: UTF-8. It is a variable-length encoding that preserves ASCII transparency. UTF-8 usage is fully conformant with the Unicode Standard and ISO/IEC 10646.

Existing software and practices in information technology frequently depend on character data being represented as a sequence of bytes. To use Unicode character data in such systems, Unicode character values and pairs of surrogates must be transformed into a sequence of one or more bytes that represent the same information, but which are restricted in their numerical range.

UTF-8 is a variable-length encoding of byte sequences, where the high bits indicate the part of the sequence to which a byte belongs. *Table 3-1* on page 47 shows how the bits in a Unicode value (or a surrogate pair) are distributed among the bytes in the UTF-8 encoding. Any byte sequence that does not follow this pattern is an ill-formed byte sequence. See *Section 3.8, Transformations*, for a definition of ill-formed byte sequences. The UTF-8 encoding form maintains transparency for all of the ASCII code values (0..127). Furthermore, the values 0..127 do not appear in any byte of a transformed result except as the direct representation of these ASCII values. The ASCII range (U+0000..U+007F) is represented by single bytes; many of the non-ideographic scripts are represented by two bytes; the remaining Unicode scalar values are represented as three bytes; and surrogate pairs require four bytes.

Other important characteristics of UTF-8 are as follows:

- It is efficient to convert to and from 16-bit Unicode text.
- The first byte indicates the number of bytes to follow in a multibyte sequence, allowing for efficient forward parsing.
- It is efficient to find the start of a character when beginning from an arbitrary location in a byte stream. Programs need to search at most four bytes backward, and it is a simple task to recognize an initial byte.
- UTF-8 is reasonably compact in terms of the number of bytes used for encoding.
- A binary sort of UTF-8 strings gives the same ordering as a binary sort of Unicode scalar values.
- If surrogates are not used, a binary sort of UTF-8 strings gives the same ordering as the binary sort of their UTF-16 counterparts.

Sample code for transforming Unicode data (UTF-16) into UTF-8 is provided on the CD-ROM.

Character Encoding Schemes

A character encoding scheme consists of an encoding form plus byte serialization. Thus there are four character encoding schemes in Unicode: UTF-8, UTF-16, UTF-16LE, and UTF-16BE. Often, encoding forms and character encoding schemes have the same name.

The names of the Unicode encoding forms are the same as those used in other contexts. Notably, the IANA maintains a registry of *charset names* used on the Internet, which includes the same names as the encoding forms used here. Although the encoding forms here and the descriptions of the charsets registered with IANA are very similar, some important differences may arise in terms of the requirements for each. For more information, please see Unicode Technical Report #17, "Character Encoding Model," on the CD-ROM or the up-to-date version on the Unicode Web site.

2.4 Unicode Allocation

All code values in the Unicode Standard are equally accessible electronically; the exact assignment of character codes is of minor consequence for information processing. Nevertheless, for the convenience of people who will use them, the codes are grouped by linguistic and functional categories. Such grouping offers the additional benefit of supporting various space-saving techniques in actual implementations.

Allocation Areas

Figure 2-7 provides an overview of the Unicode codespace allocation. For convenience, the Unicode Standard codespace is divided into several areas, which are then subdivided into character blocks:

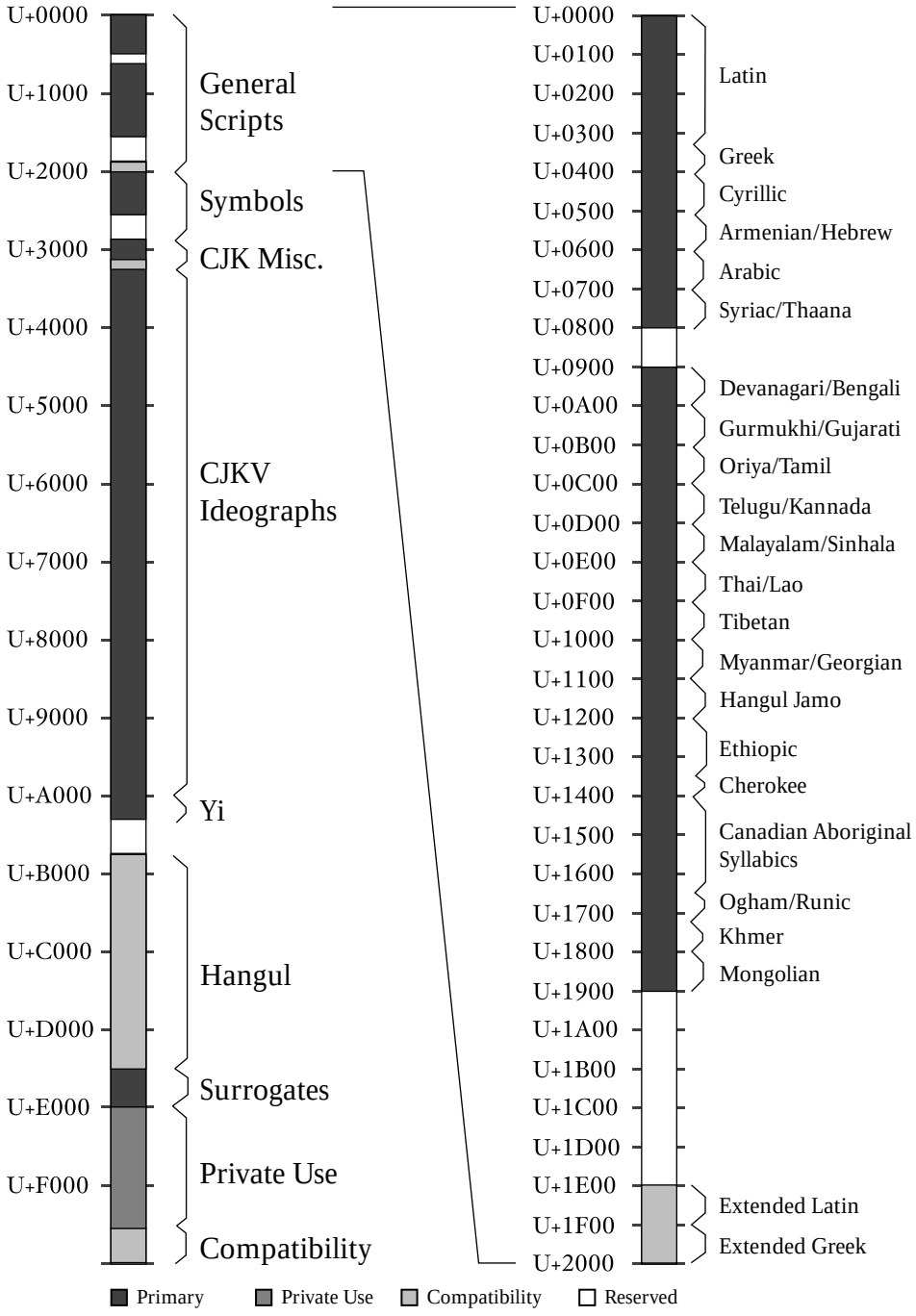
- The *General Scripts Area*, consisting of alphabetic and syllabic scripts that have relatively small character sets, such as Latin, Cyrillic, Greek, Hebrew, Arabic, Devanagari, and Thai
- The *Symbols Area*, including a large variety of symbols and dingbats, for punctuation, mathematics, chemistry, technical, and other specialized usage
- The *CJK Phonetics and Symbols Area*, including punctuation, symbols, radicals, and phonetics for Chinese, Japanese, and Korean
- The *CJK Ideographs Area*, consisting of 27,484 unified CJK ideographs
- The *Yi Syllables Area*, consisting of 1,165 syllables and 50 Yi radicals
- The *Hangul Syllables Area*, consisting of 11,172 precomposed Korean Hangul syllables
- The *Surrogates Area*, consisting of 1,024 low-half surrogates and 1,024 high-half surrogates that are used in the surrogate extension method to access more than 1 million codes for future expansion
- The *Private Use Area*, containing 6,400 code positions used for defining user- or vendor-specific characters
- The *Compatibility and Specials Area*, containing many of the characters from widely used corporate and national standards that have other representations in Unicode encoding, as well as several special-use characters

The allocation of characters into areas reflects the evolution of the Unicode Standard and is not intended to define the usage of characters in implementations. For example, many

characters are included in the standard solely for reasons of compatibility with other standards but are not coded in the Compatibility Area; there are many general-purpose symbols and punctuation in the CJK Phonetics and Symbols Area, while the Hangul *conjoining jamo* are in the General Scripts Area.

A Private Use Area gives the Unicode Standard the necessary flexibility and matches widespread practice in existing standards. Successful interchange requires agreement between sender and receiver regarding interpretation of private-use codes.

Figure 2-7. Unicode Allocation



Codespace Assignment for Graphic Characters

The predominant characteristics of codespace assignment in the Unicode Standard are as follows:

- 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 do not cross 128-byte boundaries or 1,024-byte boundaries.
- 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 makes no pretense to correlate character code allocation with language-dependent collation or case.
- Unified CJK ideographs are laid out in two sections, each of which is arranged according to the Han ideograph arrangement defined in *Section 10.1, Han*. This ordering is roughly based on a radical-stroke count order.

Nongraphic Characters, Reserved and Unassigned Codes

All code points except those mentioned below are reserved for graphic characters. Code points unassigned in this version of the Unicode Standard are available for assignment in later versions of the Unicode Standard to characters of any script.

- Sixty-five codes (U+0000..U+001F and U+007F..U+009F) are reserved specifically as control codes. Of the control codes, *null* (U+0000) can be used as a string terminator as in the C language, *tab* (U+0009) retains its customary meaning, and the others may be interpreted according to ISO/IEC 6429. (See *Section 2.8, Controls and Control Sequences*, and *Section 13.1, Control Codes*.)
- Two codes are not used to encode characters: U+FFFF is reserved for internal use (as a sentinel) and should not be transmitted or stored as part of plain text. U+FFFE is also reserved; its presence indicates byte-swapped Unicode data. (See *Section 13.6, Specials*.)
- A contiguous area of codes has been set aside for private use. Characters in this area will never be defined by the Unicode Standard. These codes can be freely used for characters of any purpose, but successful interchange requires an agreement between sender and receiver on their interpretation.
- In addition, 2,048 codes have been allocated for surrogates, which are used in the extension mechanism.

2.5 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 *Section 3.12, Bidirectional Behavior*, 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. 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.

Boustrophedon writing is 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 boustrophedon. Fixed texts can, however, be written in boustrophedon by using hard line breaks and directionality overrides.

2.6 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. The 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 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 "^" 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 3.10, Canonical Ordering Behavior*, but see also *Section 3.12, Bidirectional Behavior*.

In the charts for Indic scripts, some vowels are depicted to the left of dotted circles (see *Figure 2-8*). 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.

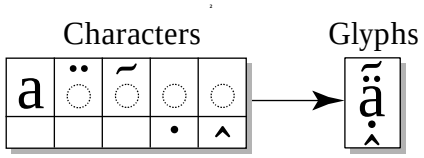
Figure 2-8. Indic Vowel Signs

Multiple Combining Characters

In some instances, more than one diacritical mark is applied to a single base character (see *Figure 2-9*). The Unicode Standard does not restrict the number of combining characters

that may follow a base character. The following discussion summarizes the treatment of multiple combining characters. (For the formal algorithm, see *Chapter 3, Conformance.*)

Figure 2-9. 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 2-10*). 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.

Figure 2-10. Interacting Combining Characters

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
ậ	LATIN SMALL LETTER A + COMBINING DOT BELOW + COMBINING DOT ABOVE LATIN SMALL LETTER A + COMBINING DOT ABOVE + COMBINING DOT BELOW
ấ̂	LATIN SMALL LETTER A WITH CIRCUMFLEX AND ACUTE LATIN SMALL LETTER A WITH CIRCUMFLEX + COMBINING ACUTE LATIN SMALL LETTER A + COMBINING CIRCUMFLEX + COMBINING ACUTE
ấ̂	LATIN SMALL LETTER A 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 combining characters override the default stacking behavior by being positioned horizontally rather than stacking or by ligature with an adjacent nonspacing mark (see *Figure 2-11*). 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 2-11*, the top example is correct and the bottom example is incorrect.

Figure 2-11. Nondefault Stacking

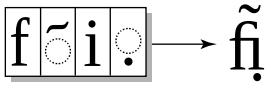
ᾴ	GREEK SMALL LETTER ALPHA + COMBINING COMMA ABOVE (psili) + COMBINING ACUTE ACCENT (oxia)	This is correct
ᾴ	GREEK SMALL LETTER ALPHA + COMBINING ACUTE ACCENT (oxia) + COMBINING COMMA ABOVE (psili)	This is incorrect

Prominent characters that show such override behavior are 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 above their base letter 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 nature of rendering combining diacritical marks.

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 2-12*). 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 5.13, Strategies for Handling Nonspacing Marks*.)

Figure 2-12. Multiple Base Characters



Multiple base characters 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 14, Code Charts.*

2.7 Special Character and Noncharacter Values

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 3.9, Special Character Properties*; for more information about such characters, see *Section 13.1, Control Codes*, *Section 13.2, Layout Controls*, and *Section 13.6, Specials.*

Byte Order Mark (BOM)

The canonical encoding form of Unicode plain text as a sequence of 16-bit codes is 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 values, 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. The *byte order mark* is not a control character that selects the byte order of the text; rather, its function is to notify recipients as to 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. 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. It is therefore not likely 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 values. 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 13.6, Specials*, for more information on *byte order mark* and its use as an encoding signature.

Special Noncharacter Values

U+FFFF and U+FFFE. These code values are *not* used to represent Unicode characters. U+FFFF is reserved for private program use as a sentinel or other signal. (Notice that U+FFFF is a 16-bit representation of –1 in two's-complement notation.) Programs receiving this code are not required to interpret it in any way. It is good practice, however, to recognize this code as a noncharacter value and to take appropriate action, such as by indicating possible corruption of the text. U+FFFE is similar in all respects to U+FFFF, except that it is also the mirror image of U+FEFF ZERO WIDTH NO-BREAK SPACE (*byte order mark*). The presence of U+FFFE constitutes a strong hint that the text in question is byte-reversed. See *Section 13.6, Specials.*

Separators

Line and Paragraph Separators. For historical reasons, carriage return and line feed are not used consistently across different systems. The Unicode Standard provides (and encourages the use of) the *line* and *paragraph separator* characters to provide clear information about where line and paragraph boundaries occur. Paragraph separation is also required for use with the bidirectional algorithm (see *Chapter 3, Conformance*). As these entities are separator codes, it is not necessary to start the first line or paragraph or to end the last line or paragraph with them. Also see *Section 13.2, Layout Controls*.

Interaction with CR/LF. The Unicode Standard does not prescribe specific semantics for U+000D CARRIAGE RETURN (CR) and U+000A LINE FEED (LF). These codes are provided to represent any CR or LF characters employed by a higher-level protocol or retained in text translated from other standards. It is left to each application to interpret these codes, to decide whether to require their use, and to determine whether CR/LF pairs or single codes are needed. See also *Section 5.9, Line Handling*, and Unicode Technical Report #13, "Unicode Newline Guidelines," on the CD-ROM or the up-to-date version on the Unicode Web site.

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 13.2, 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 values (see *Section 5.3, Unknown and Missing Characters*, and *Section 13.6, Specials*).

2.8 Controls and Control Sequences

Control Characters

The Unicode Standard provides 65 code values for the representation of control characters. These ranges are U+0000..U+001F and U+007F..U+009F, which correspond to the 8-bit controls 00_{16} to $1F_{16}$ (C0 controls) and $7F_{16}$ to $9F_{16}$ (*delete* and C1 controls). For example, the 8-bit version of *horizontal tab* (HT) is at 09_{16}; the Unicode Standard encodes *tab* at U+0009. When converting control codes from existing 8-bit text, they are merely zero-extended to the full 16 bits of Unicode characters.

Programs that conform to the Unicode Standard may treat these 16-bit control codes in exactly the same way as they treat their 7- and 8-bit equivalents in other protocols, such as ISO/IEC 2022 and ISO/IEC 6429. Such usage constitutes a higher-level protocol and is beyond the scope of the Unicode Standard. Similarly, the use of ISO/IEC 6429:1992 control sequences (extended to 16 bits) for controlling bidirectional formatting is a legitimate higher-level protocol layered on top of the plain text of the Unicode Standard. As with all higher-level protocols, both the sender and the receiver must agree upon a common protocol beforehand.

Escape Sequences. In converting text containing escape sequences to the Unicode character encoding, text must be converted to the equivalent Unicode characters. Converting escape sequences into Unicode characters on a character-by-character basis (for instance, ESC–A turns into U+001B ESCAPE, U+0041 LATIN CAPITAL LETTER A) allows the reverse conversion to be performed without forcing the conversion program to recognize the escape sequence as such.

Control Code Sequences Encoding Additional Information about Text. If a system uses sequences beginning with control codes to embed additional information about text (such as formatting attributes or structure), then such sequences form a higher-level protocol outside the scope of the Unicode Standard. Such higher-level protocols are not specified by the Unicode Standard; their existence cannot be assumed without a separate agreement between the parties interchanging such data.

Representing Control Sequences

Control sequences can be represented in the Unicode encoding design but must then be represented in terms of 16-bit characters. For example, suppose that an application allows embedded font information to be transmitted by means of an 8-bit sequence. In the following, the notation **^A** refers to the C0 control code 01_{16}, **^B** refers to the C0 control code 02_{16}, and so on:

^ATimes**^B** = **01**,54,69,6D,65,73,**02**

Then the corresponding sequence of Unicode character codes would be

^ATimes**^B** = **0001**,0054,0069,006D,0065,0073,**0002**

That is, each Unicode character code is a 16-bit zero-extended code value of the corresponding 8-bit code value.

Where the embedded data are not interpreted as a sequence of characters by the protocol, the information could be encoded as follows:

^ATimes**^B** = **0001**,5469,6D65,7300,**0002**

The data could never be encoded as

^ATimes**^B** = **01**54,696D,6573,**02**00

because in the Unicode character encoding this sequence represents four characters—LATIN CAPITAL LETTER R ACUTE (U+0154), two Han characters (U+696D and U+6573, respectively), and LATIN CAPITAL LETTER A WITH DOUBLE GRAVE (U+0200). None of these characters is a control character. If a control sequence contains embedded binary data, then the data bytes do not necessarily need to be zero-extended because the control sequence constitutes a higher protocol. However, doing so allows code conversion algorithms to succeed even in the absence of explicit knowledge of employed control sequences.

2.9 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. The following 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 as 16-bit units.

  U+2020 (that is, 2020_{16}) is the single Unicode character DAGGER '†', *not* two ASCII spaces.

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

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

  U+00D4 'Ô' is equivalent to U+004F 'o' followed by U+0302 '◌̂', but *not equivalent to* U+0302 followed by U+004F.

  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 *Section 3.12, Bidirectional Behavior*.

  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 characters in the Private Use Area.

Examples of characters that might be encoded in the Private Use Area include supplementary 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 from other standards to the Unicode Standard will be considered conformant if the matching table produces accurate conversions in both directions.

Characters Not Used in a Subset

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 this subset. Furthermore, this subset is typically different for different aspects of an implementation. For example, an application may be able to read, write, and store any 16-bit 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, uninterpretable characters must not be removed or altered. (See also *Section 3.1, Conformance Requirements.*)

2.10 Referencing Versions of the Unicode Standard

For most character encodings, the character repertoire is fixed (and often small). Once the repertoire is decided upon, it is never changed. Addition of a new abstract character to a given repertoire is conceived of as creating a new repertoire, which then will be given its own catalog number, constituting a new object.

For the Unicode Standard, on the other hand, the repertoire is inherently open. Because Unicode is a universal encoding, any abstract character that could ever be encoded is potentially a member of the actual set to be encoded, regardless of whether the character is currently known.

Each new version of the Unicode Standard replaces the previous one and makes it obsolete, but implementations—and more significantly, data—are not updated instantly. In general, major and minor version changes include new characters, which do not create particular problems with old data. The Unicode Technical Committee will neither remove nor move characters, but characters may be deprecated. This approach does not remove them from the standard or from existing data. The code point will never be used for a different character, but its use is strongly discouraged.

Implementations should be prepared to be forward-compatible with respect to Unicode versions. That is, they should accept text that may be expressed in future versions of this

standard, recognizing that new characters may be assigned in those versions. Thus they should handle incoming unassigned code points as they do unsupported characters. (See *Section 5.3, Unknown and Missing Characters*.)

A version change may also involve changes to the properties of existing characters. When this situation occurs, modifications are made to UnicodeData.txt and other relevant contributing data files, and a new update version is issued for the standard. Changes to the data files may alter program behavior that depends on them.

Version Numbering. Version numbers for the standard consist of three fields: the major version, the minor version, and the update version. The differences among them are as follows:

- Major—significant additions to the standard, published as a book.
- Minor—character additions or more significant normative changes, published as a technical report on the Unicode Web site.
- Update—any other changes to normative or important informative portions of the standard that could change program behavior. These changes are reflected in a new UnicodeData.txt file and other contributing data files of the Unicode Character Database.

For additional information on the current and past versions of the Unicode standard, see http://www.unicode.org/unicode/standard/versions/ on the Unicode Web site.

This PDF file is an excerpt from *The Unicode Standard, Version 3.0*, issued by the Unicode Consortium and published by Addison-Wesley. The material has been modified slightly for this online edition, however the PDF files have not been modified to reflect the corrections found on the Updates and Errata page (see http://www.unicode.org/unicode/uni2errata/UnicodeErrata.html). More recent versions of the Unicode standard exist (see http://www.unicode.org/unicode/standard/versions/).

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

ISBN 0-201-61633-5

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

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

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

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

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

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

First printing, January 2000.