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 which convey the essence of the standard. The Unicode Design Principles can serve as a kind of tutorial framework for understanding the Unicode Standard, and are a useful place to start from 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 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.

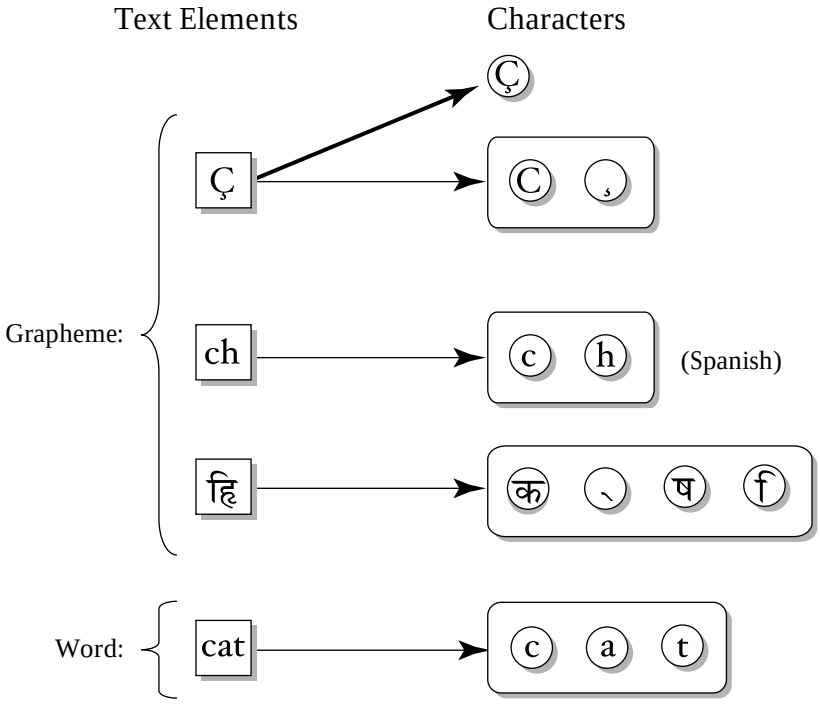
However, a character encoding standard provides just the fundamental units of encoding (that is, the abstract characters), which must exist in a unique relationship to the assigned numerical *code points*. Assigned characters are the smallest interpretable units of stored text.

An important class of text elements is called a *grapheme cluster*, which typically corresponds to what a user thinks of as a “character.” A precise definition of this concept can be found in Unicode Standard Annex #29, “Text Boundaries.” *Figure 2-1* illustrates the relationship between abstract characters and grapheme clusters.

[errata email from Mark on 8-24-00 had corrected figure for Devanagari grapheme in Fig 2-1-- still need to correct first glyph in third column.]

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*, in *Chapter 8, Middle Eastern Scripts*, and *Section 9.1, Devanagari*, in *Chapter 9, South 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 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 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.

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, obsoletes 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. By contrast, the Unicode approach assumes that characters and their properties are inherently and inextricably associated. All levels of Unicode implementations 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. G*lyphs* 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*.

[Mark to rewrite the following two paras and Fig 2-2 with material from “Where is my Character?” 8-18-02]

Unicode characters represent primarily, but not exclusively, the letters, punctuation, and other signs that constitute natural language text and technical notation. Characters are rep-

resented 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.

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.

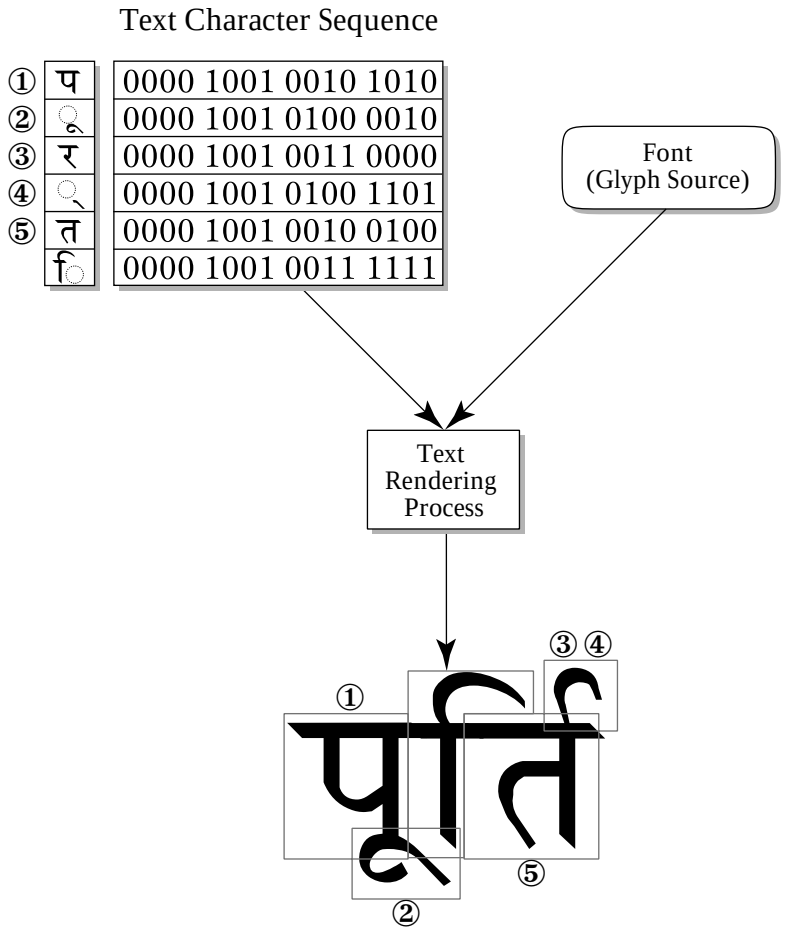
The process of mapping from characters in the memory representation to glyphs is one aspect of text rendering. The final appearance of rendered text may also depend on context (neighboring characters in the memory representation), variations in typographic design of the fonts used, and formatting information (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*.

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. The properties identified by the Unicode Standard include numeric, spacing, combination, and directionality properties (see *Chapter 4, Character Properties*). Additional

Figure 2-3. Unicode Character Code to Rendered Glyphs



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.

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

The simplicity of plain text gives it a natural role as a major structural element of rich text. SGML, RTF, HTML, XML, or 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. 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.

[Kern TEX above. 8-18-02]

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.
- 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 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 languages, 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 or *tags* from the "real" content.

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:

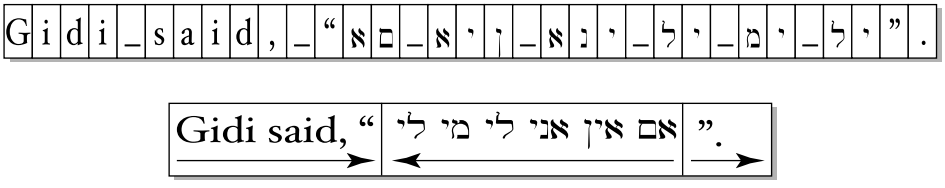
> *Plain text must contain enough information to permit the text to be rendered legibly, and nothing more.*

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.

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 2-4*.

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. Unicode Standard Annex #9, "The Bidirectional Algorithm," provides rules for the correct presentation of text containing left-to-right and right-to-left scripts.

Note that there are circumstances where the implicit bidirectional ordering will not suffice to produce comprehensible text. To deal with these cases, a minimal set of directional formatting codes is defined in the Unicode Standard to control the ordering of characters when rendered. 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 "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 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. This is important in order to avoid visual ambiguity.

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  .

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 which could have been unified with existing visually similar Unicode characters, or which 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

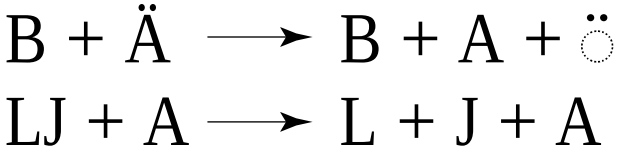
The Unicode Standard allows for the dynamic composition of accented forms and Hangul syllables. Combining characters used to create composite forms are productive. Because the process of character composition is open-ended, new forms with modifying marks may be created from a combination of base characters 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 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 3.7, 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 2-17*).

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. Any distinctions made between nonidentical equivalent sequences by applications or users are not guaranteed to be interchangeable; such distinctions must be avoided wherever possible. Where uniqueness is required, a normalized form of Unicode can be used. There are two forms that are defined. These are known as Normalization Form D (NFD) and Normalization Form C (NFC). Roughly speaking, NFD decomposes characters where possible, while NFC composes characters where possible. For more information, see Unicode Standard Annex #15, "Unicode Normalization Forms" and *Section 5.7, 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:

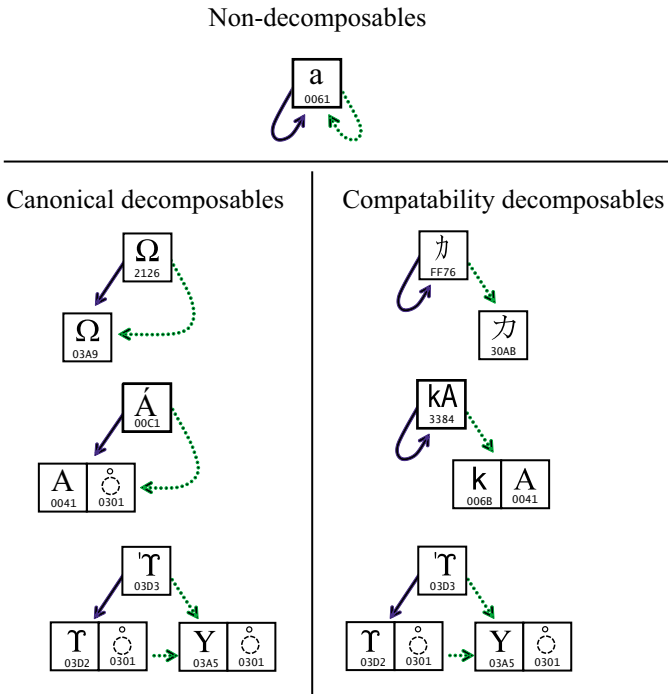
- *Canonical.* The character and its decomposition should be treated as essentially equivalent.
- *Compatibility.* The decomposition may remove some information (typically formatting information) that is important to preserve in particular contexts. By definition, compatibility decomposition is a superset 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 2-6 illustrates these three types.

Figure 2-6. Types of Decomposables



The solid arrows indicate canonical decompositions, and the dotted arrows indicate compatibility decompositions. If an arrow loops back and points to the character itself, that indicates that there is no decomposition of that type (other than in the trivial sense of a character "decomposing" to itself).The figure illustrates two important things to keep in mind:

- 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*.

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) because they are in some sense 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.

There is an area called the Compatibility and Specials Area, which 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, which is strictly defined in the conformance clauses: 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 been referred to ambiguously simply as compatibility characters. 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 composite 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 the characters to be encoded and to establish systematic rules for how the numbers represent the characters.

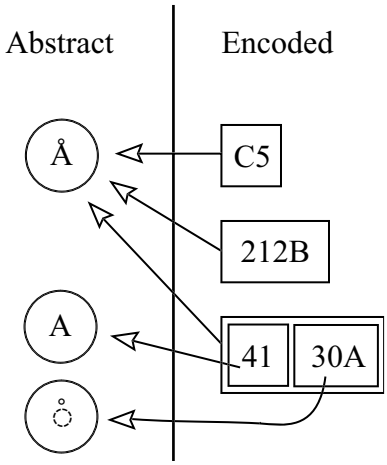
The range of integers used to code the abstract characters is called the *codespace*. A particular integer in this set is called a *code point*. When an abstract character is mapped or *assigned* to a particular code point in the codespace, it is then referred to as an *encoded character.*

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 2.7, Unicode Allocation.*

Figure 2-7 illustrates the relationship between abstract characters and code points, which together constitute encoded characters. Note that some abstract characters may be associated with more than one character (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. The hollow arrow shows a case where an encoded character sequence represents an abstract character, but does directly encode it.

Figure 2-7. 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 point, but to prevent ambiguity, the official Unicode name of the character is often also added; this clearly identifies the abstract character which is encoded. Thus, 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 which 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.

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 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 2.11, Controls and Control Sequences*, and *Section 15.1, Control Codes.*)

Table 2-2. Types of Code Points

<table>
<tr><th></th><th>Basic Type</th><th>Brief Description</th><th>General Category</th><th>Character Status</th><th>Code Point Status</th></tr>
<tr><td>1</td><td>Graphic</td><td>letter, mark, number, punctuation, symbol, and spaces</td><td>L, M, N, P, S, Zs</td><td rowspan="4">Assigned to abstract character</td><td rowspan="6">Designated (Assigned) code point</td></tr>
<tr><td>2</td><td>Format</td><td>invisible but affects neighboring characters; includes line/paragraph separators</td><td>Cf, Zl, Zp</td></tr>
<tr><td>3</td><td>Control</td><td>usage defined by protocols or standards outside the Unicode Standard</td><td>Cc</td></tr>
<tr><td>4</td><td>Private-use</td><td>usage defined by private agreement outside the Unicode Standard</td><td>Co</td></tr>
<tr><td>5</td><td>Surrogate</td><td>permanently reserved for UTF-16; restricted interchange</td><td>Cs</td><td rowspan="2">Not assigned to abstract character</td></tr>
<tr><td>6</td><td>Noncharacter</td><td>permanently reserved for internal usage; restricted interchange</td><td rowspan="2">Cn</td></tr>
<tr><td>7</td><td>Reserved</td><td>reserved for future assignment; restricted interchange</td><td>Undesignated (Unassigned) code point</td></tr>
</table>

Noncharacters. Sixty-six 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. Other noncharacters include U+FDD0..U+FDEF and the last two code points on each plane. (See *Section 15.8, Noncharacters*.)

Private Use. Three ranges of codes have been set aside for private use. Characters in these areas 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. (See *Section 15.7, Private Use Characters*.)

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

Restricted Interchange. Code points which 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 only 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.
- Reserved code points that are reserved in the latest version of the Unicode Standard cannot be interchanged. However, 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 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; thus they 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 the standard; the exceptions are Controls, Private-use and Noncharacters. While the semantics of some common Control characters (Tab, CR, LF, FF, NEL) are specified by the Unicode standard, most Control characters have semantics determined by other standards or protocols (such as ISO/IEC 6429). The semantics of private-use characters are 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 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 which 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 a sequence that contains the same code units.

To illustrate the problems with overlapping encodings, see *Figure 2-8*. In this encoding (Windows code page 932), characters are formed from either one or two code bytes. Whether a sequence is one or two 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 searchs 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 backwards through text.

The situation is made more complex by the fact that lead and trail bytes can also overlap, as in the second part of *Figure 2-8*. This means that the scan backwards 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 2-9*. This is not only inefficient, it is extremely error-prone: corruption of one byte can cause entire lines of text to be corrupted.

Figure 2-8. Overlap in Legacy Mixed-Width Encodings

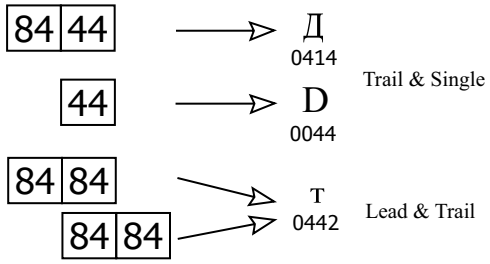
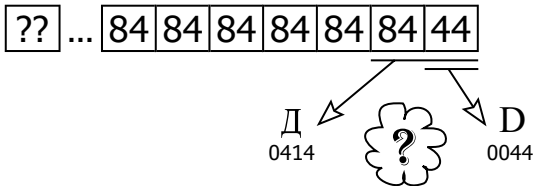


Figure 2-9. Boundaries and Interpretation



The Unicode encoding forms avoid this problem, because *none* of 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 in order to correctly determine a character boundary. This property of the encoding forms has sometimes also 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.

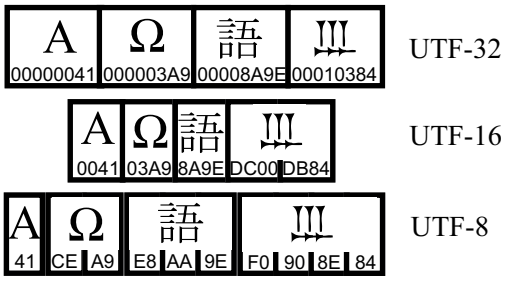
For example, when randomly accessing a string, a program can find the boundary of a character with limited backup. In UTF-16, if a pointer points to a low-surrogate, a single backup is required. In UTF-8, if a pointer points to a byte starting with 10xxxxxx (in binary), one to three backups are required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

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 2-10 shows the three Unicode encoding forms, and how they are related to Unicode scalar values.

- The dotted arrow illustrates a hypothetical private-use variant of the A-ring character (The value $F0000_{16}$ is the Unicode code point in one of the supplementary private use areas; in UTF-16 it would be represented by surrogate code units <DB80 DC00>.)

Figure 2-10. Unicode Encoding Forms



UTF-8, UTF-16, and UTF-32 are further described in the subsections which 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*.

Encoding Schemes. Whenever character data must be *serialized* into a sequence of bytes, those resulting byte sequences must be exactly defined. Machine architectures differ in *ordering* in terms of whether the most significant byte or the least significant byte comes first. These sequences are known as “big-endian” and “little-endian” orders, respectively. The details of serialization not only differ for each of the encoding forms, but for 16- and 32-bit encoding forms they also depend on whether the CPU supports big-endian or little-endian integral data. Providing the detailed specification of serialization into bytes is the function of character encoding *schemes*.

A *character encoding scheme* consists of a specified character encoding form plus a specification of how the bytes are serialized. 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. (For more on the byte order mark, see *Section 15.9, Specials*.)

For UTF-8, as for any encoding form that uses 8-bit code units, the encoding scheme consists merely of the UTF-8 code units (= bytes) in sequence. 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 7 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

*Because UTF-8 code units are 8 bits in size, 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, but may be encountered in contexts where UTF-8 data is converted from other encoding forms which use a BOM.

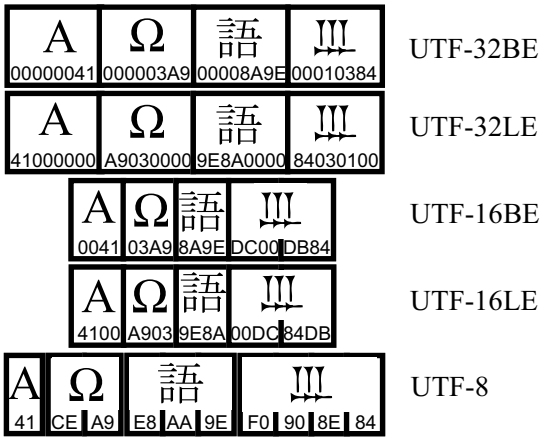
[Need to fix table footnote style above; fix widow/orphan on table 9-29-02]

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 be determinable somehow.

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 2-11 illustrates the Unicode character encoding schemes, showing how each is derived from one of the encoding forms by serialization of bytes.

Figure 2-11. Unicode Encoding Schemes



In *Figure 2-11*, the columns labeled "Serialized" show how the encoded character or encoded character sequence would be serialized in each of two different Unicode encoding schemes. These representations just show sequences of byte values, rather than encoded characters.

For the detailed formal definition of the Unicode encoding schemes and conformance requirements, see *Section 3.9, Unicode Encoding Forms*. 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 conversion, see Unicode Technical Report #22, "Character Mapping Markup Language."

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 truly fixed-width character encoding form.

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 precisely matches the range of characters defined for other standards such as XML, and also guarantees interoperability with the UTF-16 encoding form.

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 are no particular concern, but where fixed-width, single code unit access to characters is desired.

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. And 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 definitely 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 historic 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, however, 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 which works well with older Unicode implementations, and which 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.

Note that UTF-16 binary order 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 5.18, 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-length 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 which 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-length 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; the Unicode 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 multi-byte 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 UTFs, there is no overlap.

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, over 99% 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. Because of this, 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.

As a result, 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. UTF-16 provides for them the right mix of compact size with the ability to handle the occasional character outside the BMP.

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 user expectations of the identity of “characters” may lead to requirements that commonly implemented processes deal with inherently variable-width text elements.

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 3-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 sorted lists between UTF-8 systems and UTF-16 systems.

2.6 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 sequences, 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 only occur ephemerally, such as in dealing with keyboard events. While an ideal protocol would allow keyboard events to contain complete strings, many only allow 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 converted into a Unicode encoding form—even one with the same code unit size—the resulting Unicode encoding form must not violate the requirements of that encoding form. For example, isolated surrogates in a Unicode 16-bit string must not carry over into UTF-16. There are a number of different techniques for handling such conversion: omitting the isolated surrogate, converting it into U+FFFD REPLACEMENT CHARACTER, or halting the conversion with an error. For more information on this topic, see Unicode Technical Report #22, “Character Mapping Markup Language (CharMapML).”

2.7 Unicode Allocation

For the convenience of people who use them, 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 conversion tables, character property 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, or *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.

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

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

The Supplementary Ideographic Plane (SIP, or Plane 2) is the spillover allocation area for those CJK characters which 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 historic interest only.

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

Areas and 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 necessary 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 eight different blocks: Basic Latin, Latin-1 Supplement, Latin Extended-A, Latin Extended-B, IPA 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 Technical Report #24, "Script Names," and Unicode Technical Report #18, "Unicode Regular Expression Guidelines."

Details of Allocation

Figure 2-12 gives an overall picture of the allocation areas of the Unicode Standard, with an emphasis on the identity of the planes.

In *Figure 2-12*, Plane 2 and Plane 14 are shown in expanded form, to illustrate their allocation substructures. 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, towards the end of the plane, dedicated to additional CJK compatibility ideographic characters—which are basically just duplicated character encodings required for roundtrip conversion to various existing legacy East Asian character sets. The CJK compatibility ideographic characters in Plane 2 are currently all dedicated to roundtrip 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 roundtrip 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 2-12 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 6400 private use characters located in the BMP.

[There are blank pages preceding each of the full page figures 2-12, 2-13 and 2-14. These will be fixed prior to publication. JDA 10-11-02]

Figure 2-12. Unicode Allocation

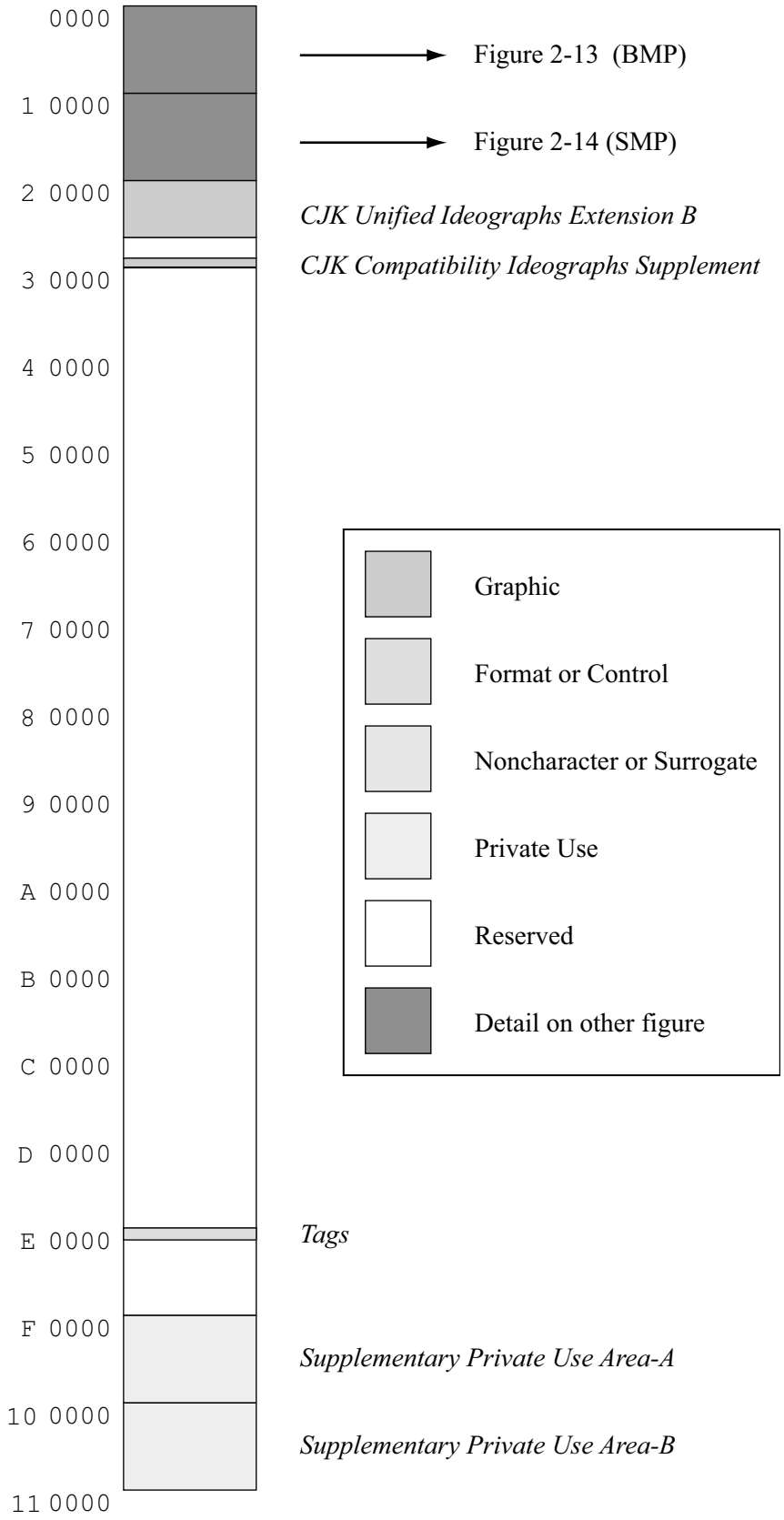
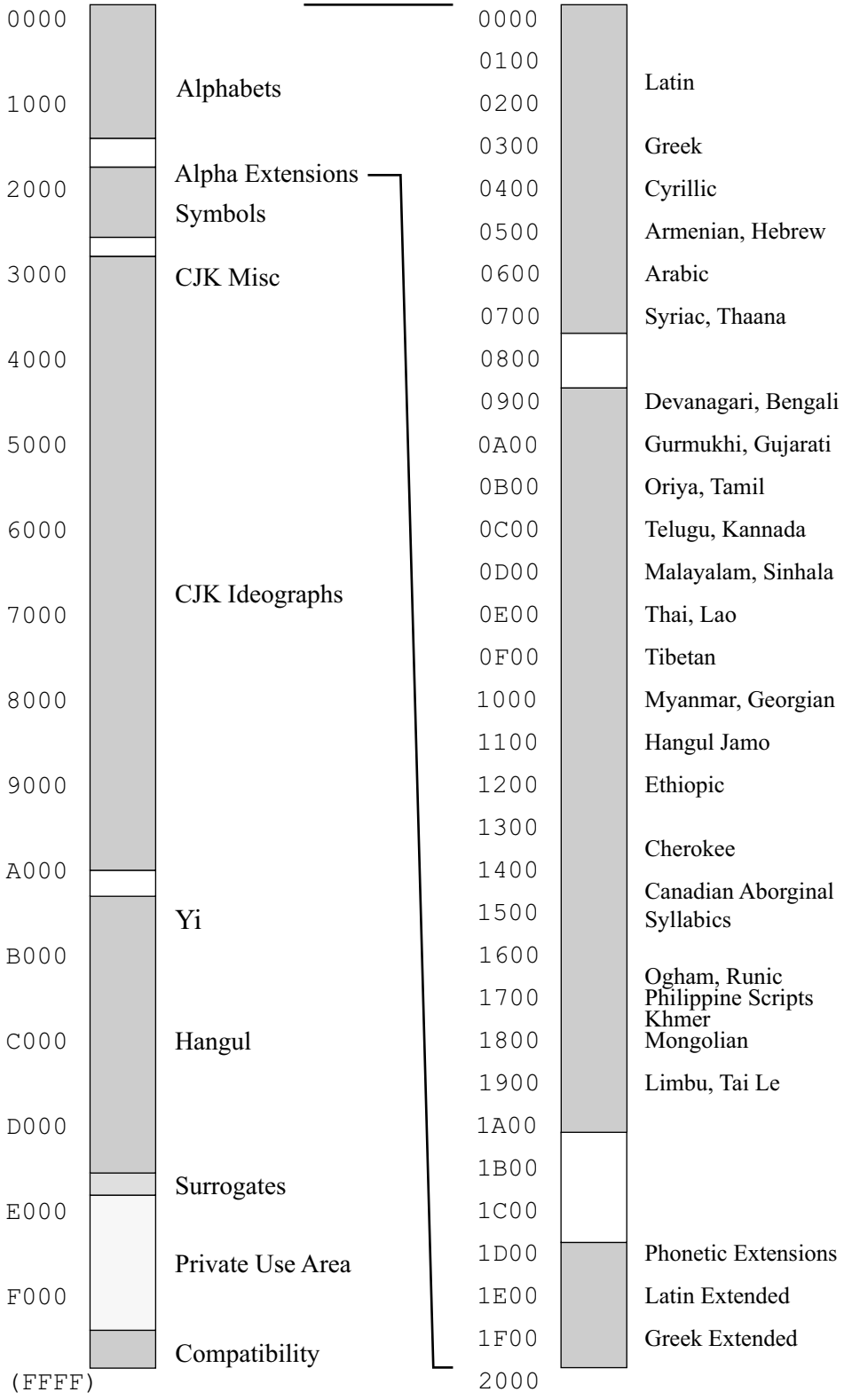


Figure 2-13 shows the BMP in an expanded format to illustrate the allocation substructure of that most important plane in more detail.

Figure 2-13. 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 2-13*. The order of the various scripts can serve as a guide to the relative position where these scripts are found in the code charts. Most of the characters encoded in this area are graphic characters, but all 65 control characters 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. This 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 a CJK Miscellaneous Area. This 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. This 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 Vertical Extension A (an additional 6,582 unified Han ideographs).

An 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.

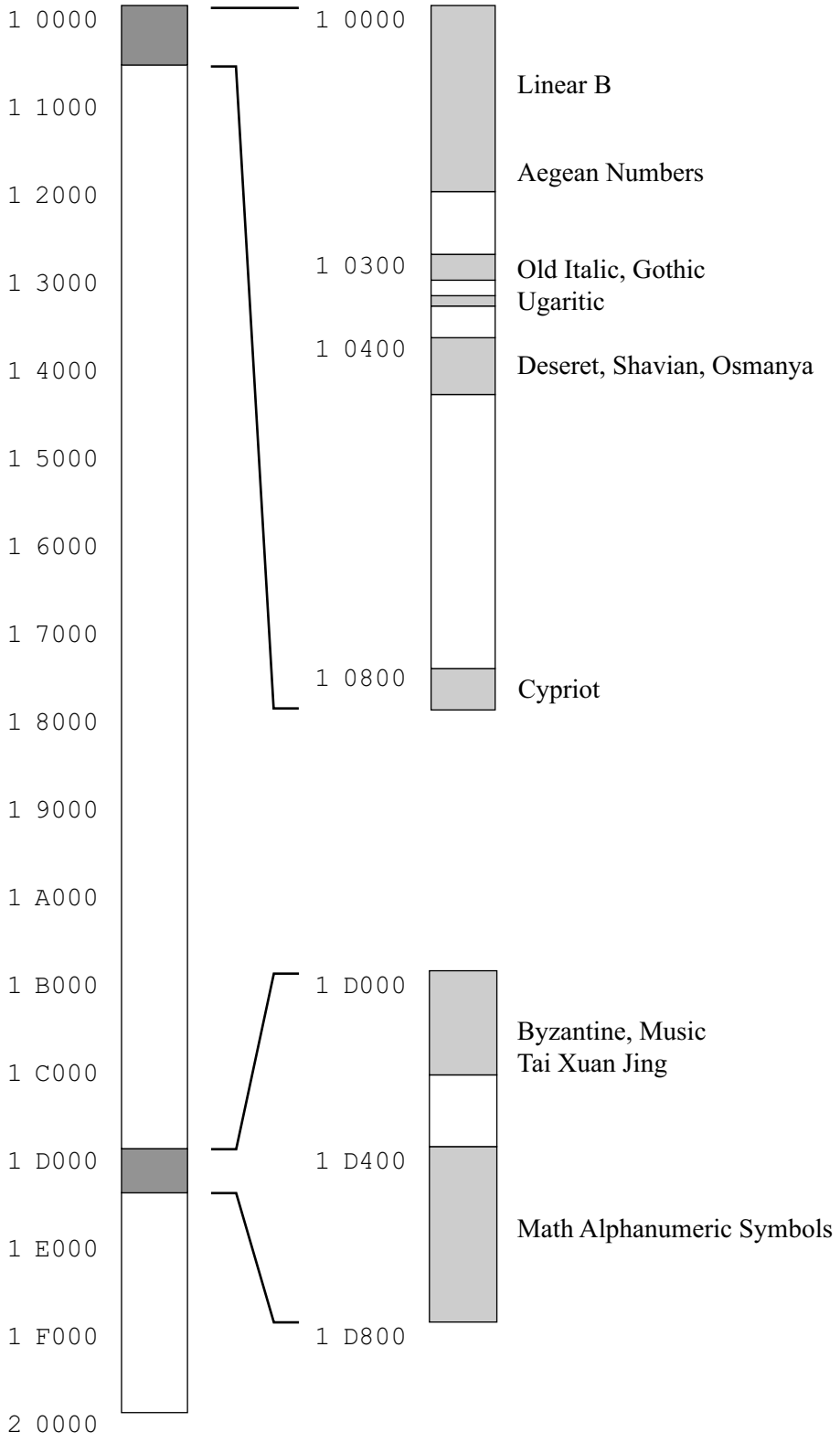
The Private Use Area in the BMP contains 6,400 private-use characters.

Finally, at the very end of the BMP, there is a 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 amongst 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.

The detailed allocation of Plane 1 is shown in *Figure 2-14*.

Figure 2-14. Allocation on Plane 1



Plane 1 currently only has 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 do not cross even 128-code-point boundaries or even 1,024-code-point 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 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.

2.8 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. 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.

2.9 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 3.11, Canonical Ordering Behavior*, but see also Unicode Standard Annex #9, "The Bidirectional Algorithm."

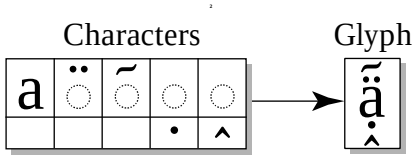
In the charts for Indic scripts, some vowels are depicted to the left of dotted circles (see *Figure 2-15*). This special case must be carefully distinguished from that of general combining diacritical mark characters. Such vowel signs are rendered to the left of a consonant letter or consonant cluster, 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 2-15. Indic Vowel Signs

Multiple Combining Characters

In some instances, more than one diacritical mark is applied to a single base character (see *Figure 2-16*). 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 2-16. 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 2-17*). 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 2-17. Interaction of Combining Characters

ã	LATIN SMALL LETTER A WITH TILDE LATIN SMALL LETTER A + COMBINING TILDE
ȧ	LATIN SMALL LETTER A + COMBINING DOT ABOVE
ą̃	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
ấ	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 2-18*). 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 2-18*, the top example is correct and the bottom example is incorrect.

Figure 2-18. Nondefault Stacking

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

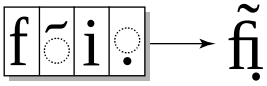
Prominent characters that show such override behavior are associated with specific scripts or alphabets. For example, when used with the Greek script, the “breathing marks”

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 2-19*). 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 2-19. Multiple Base Characters



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

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 16, Code Charts*.

2.10 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 3.11, Canonical Ordering Behavior*; for more information about such characters, see *Section 15.1, Control Codes*, *Section 15.2, Layout Controls*, 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. 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 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 *byte order mark* and its use as an encoding signature.

Special Noncharacter Values

The Unicode Standard contains a number of code points which 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 values (see *Section 5.3, Unknown and Missing Characters*, and *Section 15.9, Specials*).

2.11 Controls and Control Sequences

Control Characters

The Unicode Standard provides 65 code points 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 generate the Unicode value of the characters.

Programs that conform to the Unicode Standard may treat these 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 (represented in one of the three Unicode encoding forms) 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.

The Unicode Standard provides specific guidelines for the handling of control characters which affect line breaking. See *Section 5.9, Newline Guidelines* for more information.

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 the Unicode encoding forms. 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 code units in UTF-16 would be

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

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

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.

2.12 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. The following section gives examples of conformance and non-conformance 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* 'ゝ' *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 standards and the Unicode Standard will be considered conformant if the conversion is accurate 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 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 3.1, Conformance Requirements.*)