

Chapter 2

General Structure

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

2.1 Architectural Context

A character code standard such as the Unicode Standard enables the implementation of useful processes operating on textual data. The interesting end products are not the character codes but the text processes, since these directly serve the needs of a system's users. Character codes are like nuts and bolts — minor, but essential and ubiquitous components used in many different ways in the construction of computer software systems. No single design of a character set can be optimal for all uses, so the architecture of the Unicode Standard strikes a balance among several competing requirements.

Basic Text Processes

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

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

- Treating text as bulk data for operations such as compressing and decompressing, truncating, transmitting, and receiving

Text Elements, Code Elements, and Text Processes

One of the more profound challenges in designing a world-wide 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 German, the letter combination “ck” is a text element for the process of hyphenation (where it appears as “k-k”), but not for the process of sorting; in Spanish, the combination “ll” may be a text element for the traditional process of sorting (where it is sorted between “l” and “m”), but not for the process of rendering; and in English, the objects “A” and “a” are usually distinct text elements for the process of rendering, but generally not distinct for the process of spell-checking. 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.

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

The design of the character encoding must provide precisely the set of code elements that allow programmers to design applications capable of implementing a variety of text processes in the desired languages. These code elements may not map directly to any particular set of text elements that are used by one of these processes.

Text Processes and Encoding

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

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

- Many assumptions about character rendering that hold true for English fail for other writing systems. Unlike English, characters in other writing systems are not necessarily rendered visible one by one in rectangles from left to right. In many cases, character positioning is quite complex and does not proceed in a linear fashion. (See the Arabic and Devanagari subsections in *Section 6.1, General Scripts Area* for detailed examples of this.)
- It is not always obvious that one set of text characters is an optimal encoding for a given language. For example, there exist two approaches 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/IEC 6937 and ISO/IEC 5426 represent them by composition instead. In the Swedish language both of these 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 encodings 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 causes some text processes, such as rendering, to be carried out more easily, but other processes, such as comparison, to be more difficult. A different encoding design for English, such as case-shift control codes, would have had the opposite effect. In designing a new encoding 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. In general, culturally expected sorting orders require arbitrarily complex sorting algorithms. The expected sort sequence for the same characters differs across languages; thus, in general, no single acceptable lexicographic ordering exists. (See *Section 5.15, Sorting and Searching* for implementation guidelines.)

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

2.2 Unicode Design Principles

The design of the Unicode Standard reflects the following ten fundamental principles (see 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 Ten Unicode Design Principles

Principle	Statement
Sixteen-bit characters	Unicode character codes have a uniform width of 16 bits.
Full encoding	The full 16-bit codespace is available to encode characters.
Characters, not glyphs	The Unicode Standard encodes characters, not glyphs.
Semantics	Characters have well-defined semantics.
Plain text	The Unicode Standard encodes plain text.
Logical order	The default for memory representation is logical order.
Unification	The Unicode Standard unifies duplicate characters within scripts across languages.
Dynamic composition	The Unicode Standard allows for the dynamic composition of accented forms.
Equivalent sequence	For static precomposed forms, the Unicode Standard provides a mapping to the equivalent dynamically composed sequence of characters.
Convertibility	Accurate convertibility is guaranteed between the Unicode Standard and other widely accepted standards.

Sixteen-Bit Characters

Unicode character codes have a uniform width of 16 bits. Plain Unicode text consists of pure 16-bit Unicode character sequences. For compatibility with existing environments,

two lossless transformations for converting 16-bit Unicode values into forms appropriate for 8- or 7-bit environments have been defined:

- UTF-8 (UCS Transformation Format-8) is the standard method for transforming Unicode values into a sequence of 8-bit codes. UTF-8 is not intended to replace the base 16-bit form of Unicode encoding but may be used where needed; for example, when transmitting data through 8-bit oriented protocols.
- UTF-7 (UCS Transformation Format-7) is the standard interchange format available for use in environments that strip the eighth bit, principally 7-bit Internet exchange. See Internet Working Group RFC-1642.

The UTF-8 and UTF-7 transformations are fully described in *Appendix A, Transformation Formats*.

Full Encoding

The full 16-bit codespace (over 65,000 code positions) is available to represent characters. (See *Section 2.3, Unicode Allocation*, on how these characters are allocated in this standard.) There are over 18,000 unassigned code positions that are available for future allocation. This number far exceeds anticipated character encoding requirements for modern and most archaic characters.

One million additional characters are accessible through the *surrogate extension mechanism*, where two 16-bit code values represent a single character. This number far exceeds anticipated character encoding requirements for all world characters and symbols.

This extension mechanism will allow implementations access to rare characters in the future. Two groups, each consisting of 1,024 code positions, are reserved for this purpose and are used in pairs to represent over 1 million additional characters. These code positions are called surrogates. In this version of the Unicode Standard, *none of these additional surrogates has been assigned*.

The surrogate mechanism is designed to coexist well with the basic form of 16-bit encoding. (See *Section 3.7, Surrogates*, for the definition of this mechanism and *Section 5.5, Handling Surrogate Characters*, for implementation guidelines.)

Characters, Not Glyphs

The Unicode Standard draws a distinction between *characters*, which are the smallest components of written language that have semantic value, and *glyphs*, which represent the shapes that characters can have when they are rendered or displayed. There are various relationships between character and glyph: a single glyph may correspond to a single character, or to a number of characters, or multiple glyphs may result from a single character. The distinction between characters and glyphs is illustrated in Figure 2-1.

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

In contrast to characters, glyphs appear on the screen or paper as particular representations of one or more characters. A repertoire of glyphs comprises 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 2-1. Characters Versus Glyphs

Glyph	Unicode Character(s)
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 LATIN SMALL LETTER F + U+0069 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, while others may be rendering forms that do not directly correspond to any one 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 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 (see Figure 2-2).

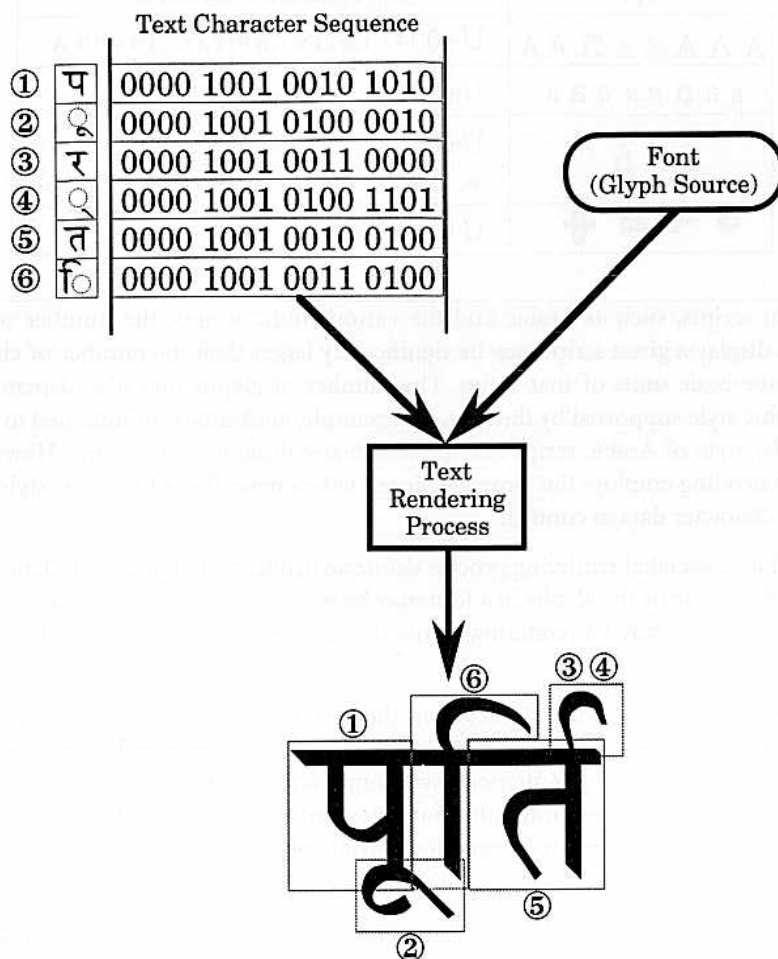
For all scripts there is an archetypical relation between character code sequences and resulting glyphic appearance. For the Latin script this is simple and well known; for several other scripts it is documented in this standard. However, in all cases, fine typography requires a more elaborated set of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

Semantics

Characters have well-defined semantics. Character property tables are provided for use in parsing, sorting, and other algorithms requiring semantic knowledge about the code points. See *Section 5.13, Locating Text Element Boundaries*, *Section 5.14, Identifiers*, and *Section 5.15, Sorting and Searching* for suggested implementations. The properties identified by the Unicode Standard include numeric, spacing, combination, and directionality properties (see *Chapter 4, Character Properties*). Additional properties may be defined as needed from time to time. In general, neither the character name nor its location in the code table designates its properties (but see also *Section 4.1, Case*).

Plain Text

Plain text is a pure sequence of character codes; plain Unicode-encoded text is a sequence of Unicode character codes. In contrast, *fancy text*, also known as *rich text*, is any text rep-

Figure 2-2. Unicode Character Code to Rendered Glyph

resentation consisting of plain text plus added information such as language identifier, font size, color, hypertext links, and so on. For example, the text of this book, a multifont text as formatted by a desktop publishing system, is fancy text.

There are many kinds of data structures that can be built into fancy text. To give but one example, in fancy text containing ideographs an application may store the phonetic readings of ideographs somewhere in the fancy text structure.

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

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

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

While there are fancy text formats that have been standardized or made public, the majority of fancy text designs are vehicles for particular implementations and are not necessarily

readable by other implementations. Since fancy text equals plain text plus added information, the extra information in fancy text can always be stripped away to reveal the “pure” text underneath. This operation is familiar, for example, in word processing systems that use both their own private fancy format and plain ASCII text file format as a universal, if limited, means of exchange. Thus, by default, plain text represents the *basic, interchangeable content of text*.

Since 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. All that is required from disparate rendering processes is to make the text legible according to the intended reading. Therefore, the relationship between appearance and content of plain text may be stated as follows:

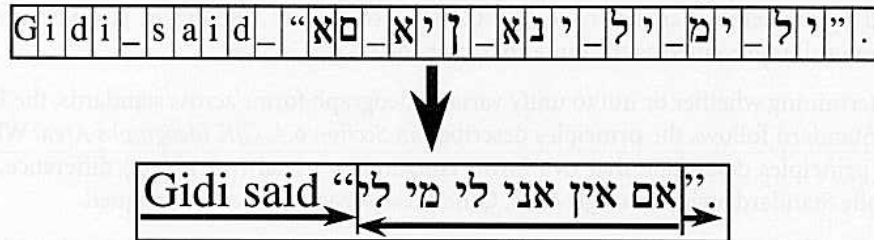
Plain text must contain enough information to permit the text to be rendered legibly, and nothing more.

The Unicode Standard encodes plain text. The distinction between data encoded in the Unicode Standard and other forms of data in the same data stream is the function of a higher-level protocol and is not specified by the Unicode Standard itself. The 64 control code positions of ISO/IEC 2022 (commonly used with ISO/IEC 646 and ISO/IEC 8859) are retained for compatibility and may be used to implement such protocols. (See *Section 2.6, Controls and Control Sequences*.)

Logical Order

For all scripts Unicode text is stored in *logical order* in the memory representation, corresponding to the order in which text is typed on the keyboard. In some circumstances the order of characters is different from this logical order when the text is displayed or printed. Where needed to ensure 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 2-3.

Figure 2-3. Bidirectional Ordering



When text in this example is ordered for display, the glyph that represents the first character of the English text is 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. This situation can arise when scripts of different directionality are mixed. The Unicode Standard includes characters to specify changes in direction. *Chapter 3, Conformance* provides rules for the correct presentation of text containing left-to-right and right-to-left scripts.

For the most part, logical order corresponds to the *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 the Devanagari subsection in *Section 6.1, General Scripts Area* for further explanation.)

Combining marks (accent marks in the Greek, Cyrillic and Latin scripts, vowel marks in Arabic and Devanagari, and so on) do not appear linearly in the final rendered text. In a Unicode character code string, all such characters *follow* the base character that they modify (for example, Roman “ä” is stored as “a” followed by combining “~” when not stored in a precomposed form). The combining marks are generally articulated in phonetic order after their base character.

Unification

The Unicode Standard avoids duplicate encoding of characters by unifying them within scripts across languages; characters that are equivalent in form are given a single code. Common letters, punctuation marks, symbols, and diacritics are given one code each, regardless of language, as are common Chinese/Japanese/Korean (CJK) ideographs. (See *Section 6.4, CJK Ideographs Area*).

Care has been taken not to make artificial distinctions among characters. Thus, for example, IPA characters are unified with the Latin alphabet. Users may become confused when they see an Å on the screen but their search dialog does not find it. The reason this occurs is that what they see on the screen is *not* an Å (*A-ring*)—it is an Å (*Ångström*). 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, duplicating characters *only* to support compatibility with base standards.

The Unicode Standard does not attempt to encode features such as language, font, size, positioning, glyphs, and so forth. For example, it does not preserve language as a part of character encoding: just as French *i grecque*, German *ypsilon*, and English *wye* are all represented by the same character code, “Y” U+0057, so too are Chinese *zi*, Japanese *ji*, and Korean *ja* all represented as the same character code, 𐤎 U+5B57.

In determining whether or not to unify variant ideograph forms across standards, the Unicode Standard follows the principles described in *Section 6.4, CJK Ideographs Area*. Where these principles determine that two forms constitute a trivial (*wazukana*) difference, the Unicode Standard assigns a single code. Otherwise, separate codes are assigned.

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

The Compatibility Area contains a large number of compatibility characters, but the Unicode Standard also contains many compatibility characters that are not in the Compatibility Area. Examples of these include Roman numerals, such as the IV “character.” By the time a distinct area for such characters was created, it was impractical to move those characters to that area. Nevertheless, it is important to be able to identify which characters are compatibility characters so that Unicode-based systems can treat them in a uniform way.

Identifying a character A as a compatibility variant of another character B implies that generally A can be remapped to B without loss of information other than formatting. Such

remapping cannot always take place because many of the compatibility characters are in place just to allow systems to maintain one-to-one mappings to existing code sets. In such cases, a remapping would lose information that is felt to be important in the original set. Compatibility mappings are called out in *Section 7.1, Character Names List*. Because replacing a character by its compatibly equivalent character or character sequence may change the information in the text, implementation has to proceed with due caution. A good use of these mappings may not be in transcoding, but in providing the correct equivalence for searching and sorting.

Dynamic Composition

The Unicode Standard allows for the dynamic composition of accented forms. 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 or any other script.

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

Equivalent Sequence

Some text elements can be encoded either as static precomposed forms or by dynamic composition. Common precomposed forms such as U+00DC LATIN CAPITAL LETTER U WITH DIAERESIS “Ü” 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.

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 2-4).

Figure 2-4. Equivalent Sequences

$$\begin{aligned} B + \ddot{A} &\rightarrow B + A + \ddot{ } \\ LJ + A &\rightarrow L + J + A \end{aligned}$$

In such cases the Unicode Standard does not prescribe one particular sequence; each of the sequences in the examples are equivalent. Systems may choose to normalize Unicode text to one particular sequence, such as normalizing composed character sequences into precomposed characters or vice-versa. Therefore, any distinctions made by applications or users are not guaranteed to be interchangeable. (For implementation guidelines see *Section 5.9, Normalization*).

Convertibility

Character identity is preserved for interchange with a number of different base standards, which included 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 guarantees that there will always be a mapping between the Unicode Standard and base standards.

Accurate convertibility is guaranteed between the Unicode Standard and other standards in wide usage as of May 1993. In general, a single code value in another standard will correspond to a single code value in the Unicode Standard. However, sometimes a single code value in another standard corresponds to a sequence of code values in the Unicode Standard, or vice versa. Conversion between Unicode text and text in other character codes must in general be done by explicit table-mapping processes. (See also *Section 5.7, Transcoding to Other Standards.*)

2.3 Unicode Allocation

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

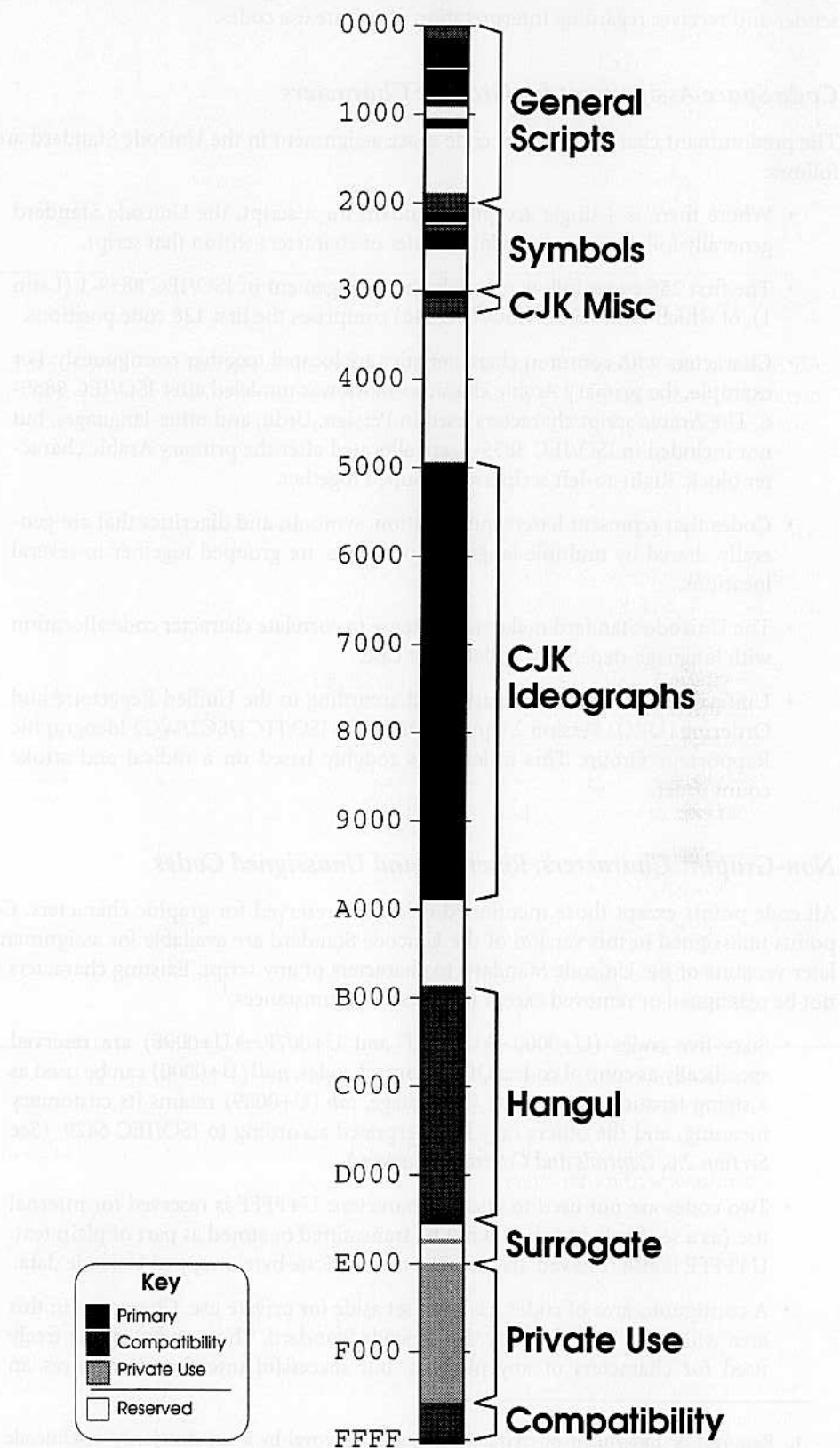
Allocation Areas

Figure 2-5 provides an overview of the Unicode code space allocation. The Unicode Standard code space is divided into several areas, which are themselves divided into character blocks.

- The *General Scripts Area*, consisting of alphabetic and syllabic scripts that have relatively small character sets, such as Latin, Cyrillic, Greek, Hebrew, Arabic, Devanagari, and Thai
- The *Symbols Area*, including a large variety of symbols and dingbats, for punctuation, mathematics, chemistry, technical, and other specialized usage
- The *CJK Phonetics and Symbols Area*, including punctuation, symbols, and phonetics for Chinese, Japanese, and Korean
- The *CJK Ideographs Area*, consisting of 20,902 unified CJK ideographs
- The *Hangul Syllables Area*, consisting of 11,172 precomposed Korean Hangul syllables
- The *Surrogates Area*, consisting of 1024 low-half surrogates and 1024 high-half surrogates that are used in the surrogate extension method to access over one million codes for future expansion
- The *Private Use Area*, containing 6,400 code positions used for defining user- or vendor-specific characters
- The *Compatibility and Specials Area*, containing characters from widely used corporate and national standards that have other representations in Unicode encoding, and several special-use characters

The allocation of characters into areas reflects the evolution of the Unicode Standard and is not intended to define the usage of characters in implementations. For example, there are many characters included in the standard solely for reasons of compatibility with other standards but not coded in the Compatibility Area; there are many general-purpose symbols and punctuation in the CJK Auxiliary Area, while the Hangul *conjoining jamo* are in the General Scripts Area.

Figure 2-5. Unicode Allocation



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

Code Space Assignment for Graphic Characters

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

- Where there is a single accepted standard for a script, the Unicode Standard generally follows it for the relative order of characters within that script.
- The first 256 codes follow precisely the arrangement of ISO/IEC 8859-1 (Latin 1), of which 7-bit ASCII (ISO/IEC 646) comprises 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.
- Codes that represent letters, punctuation, symbols, and diacritics that are generally shared by multiple languages or scripts are grouped together in several locations.
- The Unicode Standard makes no pretense to correlate character code allocation with language-dependent collation or case.
- Unified CJK ideographs are arranged according to the Unified Repertoire and Ordering (URO) Version 2.0 published by the ISO JTC1/SC2/WG2 Ideographic Rapporteur Group. This ordering is roughly based on a radical and stroke count order.

Non-Graphic Characters, Reserved and Unassigned Codes

All code points except those mentioned below are reserved for graphic characters. Code points unassigned in this version of the Unicode Standard are available for assignment in later versions of the Unicode Standard to characters of any script. Existing characters will not be reassigned or removed except in extreme circumstances.¹

- 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 2.6, *Controls and Control Sequences*.)
- Two codes are not used to encode characters: U+FFFF is reserved for internal use (as a sentinel) and should not be transmitted or stored as part of plain text. U+FFFE is also reserved. Its presence may indicate byte-swapped Unicode data.
- A contiguous area of codes has been set aside for private use. Characters in this area will never be defined by the Unicode Standard. These codes can be freely used for characters of any purpose, but successful interchange requires an

1. Removal or movement of characters requires approval by a supermajority of Unicode Consortium members. The only cases where this has happened were in the process of merging with ISO/IEC 10646 and in accommodating the addition of the full repertoire of Korean characters.

agreement between sender and receiver on their interpretation.

- 2K codes have been allocated for use in the extension mechanism surrogates. There are no escape sequences to access other code spaces, as it is not necessary to maintain state or check for escape sequences.

2.4 Special Character and Non-Character Values

Byte Order Mark

The canonical encoding form of Unicode plain text as a sequence of 16-bit codes is sensitive to the byte ordering that is used when serializing text into a sequence of bytes, such as when writing to a file or transferring across a network. Some processors place the least significant byte in the initial position, while 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 would force one class of processors to swap the byte order on reading and writing plain text files, even when the file never leaves the system on which it was created.

To have an efficient way to indicate which byte order is used in a text, the Unicode Standard contains two code values, U+FEFF ZERO WIDTH NO-BREAK SPACE (*byte order mark*) and U+FFFE (not a character code), which are the byte-ordered mirror images of each other. The *byte order mark* is not a control character that selects the byte order of the text; rather its function is to notify recipients which byte ordering is used in a file.

Unicode Signature. The sequence FE₁₆, FF₁₆ may serve as an implicit marker to identify a file as containing Unicode text. This sequence is exceedingly rare at the outset of text files using other character encodings, single- or multiple-byte.

For example, in systems that employ ISO Latin 1 (ISO/IEC 8859-1) or the Microsoft Windows ANSI Code Page 1252, this sequence constitutes the string *thorn + y umlaut* “þÿ”; in systems that employ the Apple Macintosh™ Roman character set or the Adobe Standard Encoding, this sequence represents *ogonek + hacek* “ ě”; in systems that employ other common IBM PC Code Pages (e.g., CP 437, 850, etc.), this sequence represents *black square + no-break space* “■”.

Strictly speaking, however, employment as a signature constitutes a particular use of a Unicode character, and there is nothing in this standard itself that requires or endorses this usage. Systems that employ the Unicode character encoding as their interchange code should consider prepending the U+FEFF *byte order mark* to each plain text file and removing initial *byte order marks* during processing. The *byte order mark* has legitimate use as zero width no-break space in the middle of text streams; it should not be filtered there. See the Specials subsection of *Section 6.8, Compatibility Area and Specials* for more information on the use of *byte order mark*.

Special Non-Character Values

U+FFFF and U+FFFE. These code values are *not* used to represent Unicode characters. U+FFFF is reserved for private program use as a sentinel or other signal. (Notice that U+FFFF is a 16-bit representation of -1 in two’s-complement notation.) Programs receiving this code are not required to interpret it in any way. It is good practice, however, to recognize this code as a non-character value and to take appropriate action, such as indicating possible corruption of the text. U+FFFE is similar in all respects to U+FFFF, except that it is

also the mirror image of U+FEFF ZERO WIDTH NO-BREAK SPACE (*byte order mark*). The presence of a U+FFFE constitutes a strong hint that the text in question is byte-reversed.

Separators

Line and Paragraph Separator. The Unicode Standard provides two unambiguous characters, U+2028 LINE SEPARATOR and U+2029 PARAGRAPH SEPARATOR, to separate lines and paragraphs. A new line is begun after each line separator. A new paragraph is begun after each paragraph separator. Since these are separator codes, it is not necessary either to start the first line or paragraph or to end the last line or paragraph with them. Doing so would indicate that there was an empty paragraph or line following. The paragraph separator can be inserted between paragraphs of text. Its use allows the creation of plain text files, which can be laid out on a different line width at the receiving end. The line separator can be used to indicate an unconditional end of line. These are considered the canonical form of denoting line and paragraph boundaries in Unicode plain text.

Interaction with CR/LF. The Unicode Standard does not prescribe specific semantics for U+000D CARRIAGE RETURN (CR) and U+000A LINE FEED (LF). These codes are provided to represent any CR or LF characters employed by a higher-level protocol or retained in text translated from other standards. It is left to each application to interpret these codes, to decide whether to require their use, and to determine whether CR/LF pairs or single codes are needed.

Layout and Format Control Characters

The Unicode Standard defines several characters, which are used to control joining behavior, bidirectional ordering control, and alternate 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 the General Punctuation section of *Chapter 6, Character Block Descriptions*.

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 5.4, Unknown and Missing Characters*).

2.5 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,” as “diacritic,” or as “non-spacing” 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 and not to occupy a spacing position by themselves. This is the motivation for the terms “combining” and “non-spacing.” The spacing or non-spacing properties of a combining character are really properties of the glyph used to depict a combining character, since, in certain scripts (for example, Tamil) a combining character may be depicted with either depending on the context.

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

All diacritics can be applied to any base character and are available for use with any script. There is a separate block for symbol diacritics, generally intended to be used with symbol base characters. There are additional combining characters in the blocks 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, also have combining characters indicated in the charts in relation to dotted circles to show their position relative to the base character. Many of these non-spacing marks encode vowel letters; as such they are not generally referred to as “diacritics.”

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 LATIN SMALL LETTER A “a” + U+0308 COMBINING DIAERESIS “¨” + U+0075 LATIN SMALL LETTER U “u” unambiguously encodes “äü” 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. To avoid the complication of defining and implementing combining characters on both sides of base characters, the Unicode Standard specifies that all combining characters must follow their base characters. This convention conforms to the way modern font technology handles the rendering of non-spacing 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 ISO/IEC 6937 and the bibliographic standard ISO/IEC 5426.

A sequence of base character plus one or more combining characters generally has the same properties as the base character, except for the case of enclosing diacritics that convey a symbol property. For example U+2460 CIRCLED DIGIT ONE has the same property as U+0031 DIGIT ONE followed by U+20DD COMBINING ENCLOSING CIRCLE.

Figure 2-6. Indic Vowel Signs

फ + ि → फि

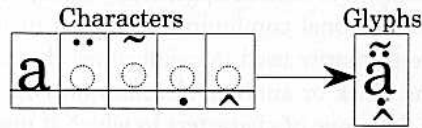
In the charts for Indian scripts, some vowels are depicted to the left of dotted circles (see Figure 2.6). This is a special case to be carefully distinguished from that of general combining diacritical mark characters. Such vowel signs are rendered to the left of a consonant letter or consonant cluster, even though their logical order in the Unicode encoding follows the consonant letter. The decision to code these in pronunciation order and not in visual order is also consistent with the ISCII standard.

Multiple Combining Characters

There are instances where more than one diacritical mark is applied to a single base character (see Figure 2-7). 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 2-7. Stacking Sequences



1. 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 2-8). 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 2-8. Interacting Combining Characters



ã	LATIN SMALL LETTER A WITH TILDE LATIN SMALL LETTER A + COMBINING TILDE
â	LATIN SMALL LETTER A + COMBINING DOT ABOVE
ãä	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
ä	LATIN SMALL LETTER A + COMBINING DOT BELOW + COMBINING DOT ABOVE LATIN SMALL LETTER A + COMBINING DOT ABOVE + COMBINING DOT BELOW
âá	LATIN SMALL LETTER A WITH CIRCUMFLEX AND ACUTE LATIN SMALL LETTER A WITH CIRCUMFLEX + COMBINING ACUTE LATIN SMALL LETTER A + COMBINING CIRCUMFLEX + COMBINING ACUTE
â	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, then tone mark character.

2. Some specific combining characters override the default stacking behavior by being positioned horizontally rather than stacking, or by ligaturing with an

adjacent non-spacing mark (see Figure 2-9). When positioned horizontally, the order of codes is reflected by positioning in the dominant order of the script with which they are used. For example, in a left-to-right script, horizontal accents would be coded left-to-right.

Figure 2-9. Overriding Behavior

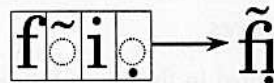
	GREEK SMALL LETTER ALPHA + COMBINING COMMA ABOVE (psili) + COMBINING ACUTE ACCENT (oxia)
	GREEK SMALL LETTER ALPHA + COMBINING ACUTE ACCENT (oxia) + COMBINING COMMA ABOVE (psili)

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 is one example of the language-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 2-10). Internally, the software has to distinguish between the non-spacing 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 non-spacing marks, see *Section 5.11, Strategies for Handling Non-Spacing Marks*.)

Figure 2-10. Multiple Base Characters



Multiple base characters do not commonly occur in most scripts. However, in some scripts, such as Arabic, this situation occurs quite often when vowel marks are used. This is 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 might be done, for example, when talking about the diacritical mark itself as a mark, rather than using it in its normal way in text. The Unicode Standard separately encodes clones of many common European diacritical marks that are spacing characters, largely to provide compatibility with existing character set standards. These related characters are cross-referenced in the names list in *Chapter 7, Code Charts*.

2.6 Controls and Control Sequences

Control Characters

The Unicode Standard provides 65 code values for the representation of control characters. These ranges are U+0000→U+001F and U+007F→U+009F, which correspond to the 8-bit controls 00₁₆ to 1F₁₆ (C0 controls) and 7F₁₆ to 9F₁₆ (*delete* and C1 controls). For example, the 8-bit version of *horizontal tab* (HT) is at 09₁₆; the Unicode Standard encodes *tab* at U+0009. When converting control codes from existing 8-bit text, they are merely zero extended to the full 16 bits of Unicode characters.

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

Escape Characters. 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 does use 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 but must then be represented in terms of 16-bit characters. For example, suppose that an application allows embedded font information to be transmitted by means of an 8-bit sequence. In the following, the notation **^A** refers to the C0 control code 01₁₆, **^B** refers to the C0 control code 02₁₆, and so on:

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

Then the corresponding sequence of Unicode character codes would be

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

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

Where the embedded data is not interpreted as a sequence of characters by the protocol, it could be encoded as:

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

The data could never be encoded as

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

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 is a control character. If a control sequence contains embedded binary data, then the data bytes do not necessarily need to be zero-extended as 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.

2.7 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 as 16-bit units.
 - U+2020 (that is, 2020_{16}) is the single Unicode character DAGGER ‘†’, *not* two ASCII spaces.
- It interprets characters according to the identities, properties, and rules defined for them in this standard.
 - U+2423 is ‘□’ OPEN BOX, *not* ‘ゝ’ hiragana small i (which is the meaning of the bytes 2423_{16} in JIS).
 - U+00D4 ‘ð’ is equivalent to U+004F ‘o’ followed by U+0302 ‘¨’, but *not* equivalent to U+0302 followed by U+004F.
 - U+05D0 ‘נ’ followed by U+05D1 ‘ב’ looks like ‘בנ’, *not* ‘נב’ when displayed.
- 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), nor to anything but U+03A1.

However, it is acceptable for that implementation:

- To support only a subset of the Unicode characters
 - An application may not provide mathematical symbols, or the Thai script.
- To transform data knowingly
 - Uppercase conversion: ‘a’ transformed to ‘A’

Romaji to kana: ‘kyo’ transformed to キヨ

247D “(10)” decomposed to 0028 0031 0030 0029

- To build higher-level protocols on the character set

Compression of characters

Use of rich text file formats

- To define characters in the Private Use Area

Examples of characters that might be encoded in the Private Use Area are supplementary ideographic characters (*gaiji*) or existing corporate logo characters.

Code conversion from other standards to the Unicode Standard will be considered conformant if the matching table produces accurate conversions in both directions.

Characters Not Used in a Subset

The Unicode Standard does not require that an application be capable of interpreting and rendering all of the Unicode characters in order 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, there is a subset of characters which an implementation is able to interpret.

The Unicode Standard provides no formalized method for identifying this subset. Furthermore, this subset is typically different for different aspects of an implementation. For example, an application may be able to read, write, and store any 16-bit character, be able to sort one subset according to the rules of one or more languages (and the rest arbitrarily), but only have access to fonts for a single script. The same implementation may be able to render additional scripts as soon as additional fonts are installed in its environment. Therefore, the subset of interpretable characters is typically not a static concept.

Conformance to the Unicode Standard *implies* that whenever text purports to be unmodified, uninterpretable characters must not be removed or altered. (See also *Section 3.1, Conformance Requirements*.)