L2/05-290

Chapter 3

Conformance

DRAFT

This chapter defines conformance to the Unicode Standard in terms of the principles and encoding architecture it embodies. The first section defines the format for referencing the Unicode Standard and Unicode properties. The second section consists of the conformance clauses, followed by sections that define more precisely the technical terms used in those clauses. The remaining sections contain the formal algorithms that are part of conformance and referenced by the conformance clause. Additional definitions and algorithms that are part of this standard can be found in the Unicode Standard Annexes listed at the end of *Section 3.2, Conformance Requirements*.

In this chapter, conformance clauses are identified with the letter *C*. Definitions are identified with the letter *D*. Bulleted items are explanatory comments regarding definitions or subclauses.

The numbering of clauses and definitions matches that of prior versions of *The Unicode Standard* where possible. Where new clauses and definitions were added, letters are used with numbers—for example, *D7a*. In a few cases, numbers have been reused for definitions.

For information on implementing best practices, see *Chapter 5, Implementation Guidelines*.

3.1 Versions of the Unicode Standard

For most character encodings, the character repertoire is fixed (and often small). Once the repertoire is decided upon, it is never changed. Addition of a new abstract character to a given repertoire creates a new repertoire, which will be treated either as an update of the existing character encoding or as a completely new character encoding.

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

Each new version of the Unicode Standard supersedes the previous one, but implementations—and more significantly, data—are not updated instantly. In general, major and minor version changes include new characters, which do not create particular problems with old data. The Unicode Technical Committee will neither remove nor move characters. Characters may be *deprecated*, but this does not remove them from the standard or from existing data. The code point for a deprecated character will never be reassigned to a different character, but the use of a deprecated character is strongly discouraged. Generally these rules make the encoded characters of a new version backward-compatible with previous versions.

Implementations should be prepared to be forward-compatible with respect to Unicode versions. That is, they should accept text that may be expressed in future versions of this standard, recognizing that new characters may be assigned in those versions. Thus, they

DRAFT

should handle incoming unassigned code points as they do unsupported characters. (See *Section 5.3, Unknown and Missing Characters*.)

A version change may also involve changes to the properties of existing characters. When this situation occurs, modifications are made to the Unicode Character Database, and a new update version is issued for the standard. Changes to the data files may alter program behavior that depends on them. However, such changes to properties and to data files are never made lightly. They are made only after careful deliberation by the Unicode Technical Committee has determined that there is an error, inconsistency, or other serious problem in the property assignments.

Stability

Each version of the Unicode Standard, once published, is absolutely stable and will *never* change. Implementations or specifications that refer to a specific version of the Unicode Standard can rely upon this stability. When implementations or specifications are upgraded to a future version of the Unicode Standard, then changes to them may be necessary. Note that even errata and corrigenda do not formally change the text of a published version; see “Errata and Corrigenda,” later in this section.

Some features of the Unicode Standard are guaranteed to be stable *across* versions. These include the names and code positions of characters, or their decompositions as well as several other character properties for which stability is important to implementations. See also “Stability of Properties” in *Section 3.5, Properties*. The details of such stability guarantees are contained in the policies on character encoding stability found on the Unicode Web site. See the subsection “Policies” in *Section B.4, Other Unicode References*. See also the discussion of backward compatibility in Unicode Standard Annex #31, “Identifier and Pattern Syntax,” and the subsection “Interacting with Downlevel Systems” in *Section 5.3, Unknown and Missing Characters*.

Version Numbering

Version numbers for the Unicode Standard consist of three fields, denoting the major version, the minor version, and the update version, respectively. For example, “Unicode 3.1.1” indicates major version 3 of the Unicode Standard, minor version 1 of Unicode 3, and update version 1 of minor version Unicode 3.1.

Formally, each new version of the Unicode Standard supersedes all earlier versions. However, because of the differences in the ways major, minor, and update versions are documented, minor and update versions generally do not obsolete all of the documentation of the immediately prior versions of the standard.

Additional information on the current and past versions of the Unicode Standard can be found on the Unicode Web site. See the subsection “Versions” in *Section B.4, Other Unicode References*. The online document contains the precise list of contributing files from the Unicode Character Database and the Unicode Standard Annexes, which are formally part of each version of the Unicode Standard.

The differences between major, minor, and update versions are as follows:

Major Version. A major version represents significant additions to the standard, including but not limited to major additions to the repertoire of encoded characters. A major version is published as a book, together with associated updates to Unicode Standard Annexes and the Unicode Character Database.

A major version consolidates all errata and corrigenda to data. The publication of the book for a major version supersedes any prior documentation for major, minor, and update versions.

DRAFT

Minor Version. A minor version also represents significant additions to the standard. It may include small or large additions to the repertoire of encoded characters or other significant normative changes. A minor version is only published online, and is not published as a book. Prior to Unicode 4.1, a minor version was published as a Unicode Standard Annex (or as a Unicode Technical Report for the very earliest minor versions). Starting with Unicode 4.1, minor versions are published as stable version pages online. A minor version is also associated with an update to the Unicode Character Database, and updates to the UAXes.

A minor version incorporates selected errata as appropriate. The documentation for a minor version does not stand alone, but rather amends the documentation of the prior version.

Update Version. An update version represents relatively small changes to the standard, focusing on updates to the data files of the Unicode Character Database. An update version never involves any additions to character repertoire. It is only published online. Starting with Unicode 3.0.1, update versions are published as stable version pages online. Prior to that version, update versions were simply documented with the list of relevant data file changes to the Unicode Character Database.

An update version incorporates selected errata, primarily for the data files. The documentation for an update version does not stand alone, but rather amends the prior version.

Errata and Corrigenda

From time to time it may be necessary to publish errata or corrigenda to the Unicode Standard. Such errata and corrigenda will be published on the Unicode Web site. See *Section 0.6, Updates and Errata* for information on how to report errors in the standard.

Errata. Errata correct errors in the text or other informative material, such as the representative glyphs in the code charts. See the subsection "Updates and Errata" in *Section B.4, Other Unicode References*. Whenever a new major version of the standard is published, all errata up to that point are incorporated into the text.

Corrigenda. Occasionally there are errors that are important enough that a corrigendum is issued prior to the next version of the Unicode Standard. Such a corrigendum does not change the contents of the previous version. Instead, it provides a mechanism for an implementation, protocol or other standard to cite the previous version of the Unicode Standard with the corrigendum applied. If a citation does not specifically mention the corrigendum, the corrigendum does not apply. For more information on citing corrigenda, see "Versions" in *Section B.4, Other Unicode References*.

References to the Unicode Standard

The documents associated with the major, minor, and update versions are called the major reference, minor reference, and update reference, respectively. For example, consider Unicode Version 3.1.1. The major reference for that version is *The Unicode Standard, Version 3.0* (ISBN 0-201-61633-5). The minor reference is Unicode Standard Annex #27, "The Unicode Standard, Version 3.1." The update reference is Unicode Version 3.1.1. The exact list of contributory files, Unicode Standard Annexes, and the Unicode Character Database can be found at Enumerated Version 3.1.1.

The reference for *this* version, Version 5.0.0, of the Unicode Standard, is:

> The Unicode Consortium. The Unicode Standard, Version 5.0.0, defined by: *The Unicode Standard, Version 5.0* (Boston, MA, Addison-Wesley, 2006. ISBN 0-NNN-NNNNN-N)

DRAFT

[Check 2006 and add in ISBN.]

References to an update or minor version include a reference to both the major version and the documents modifying it. For the standard citation format for other versions of the Unicode Standard, see "Versions" in *Section B.4, Other Unicode References*.

References to Unicode Character Properties

Properties and property values have defined names and abbreviations, such as:

Property:	General_Category (gc)
Property Value:	Uppercase_Letter (Lu)

To reference a given property and property value, these aliases are used, as in this example:

> The property value Uppercase_Letter from the General_Category property, as specified in Version 5.0.0 of the Unicode Standard.

Then cite that version of the standard, using the standard citation format that is provided for each version of the Unicode Standard.

When referencing multi-word properties or property values, it is permissible to omit the underscores in these aliases, or to replace them by spaces.

When referencing a Unicode character property it is customary to prepend the word "Unicode" to the name of the property, unless it is clear from context that the Unicode Standard is the source of the specification.

References to Unicode Algorithms

A reference to a Unicode Algorithm must specify the name of the algorithm or its abbreviation, followed by the version of the Unicode Standard, as in this example:

> The Unicode Bidirectional Algorithm, as specified in Version 4.1.0 of the Unicode Standard.
>
> See Unicode Standard Annex #9, "The Bidirectional Algorithm," (http://www.unicode.org/reports/tr9/tr9-15.html)

Where algorithms allow tailoring, the reference must state whether any such tailorings were applied or are applicable. For algorithms contained in a Unicode Standard Annex, the document itself and its location on the Unicode Web site may be cited as the location of the specification.

When referencing a Unicode algorithm it is customary to prepend the word "Unicode" to the name of the algorithm, unless it is clear from context that the Unicode Standard is the source of the specification.

Versions and Open Repertoires

Because Unicode has an open repertoire with relatively frequent updates, it is important not to over-specify the version number. Wherever the precise behavior of all Unicode characters needs to be cited, the full three-field version number should be used, as in the first example below. However, trailing zeros are often omitted, as in the second example. In such a case, writing 3.1 is in all respects equivalent to writing 3.1.0.

1. The Unicode Standard, Version 3.1.1
2. The Unicode Standard, Version 3.1
3. The Unicode Standard, Version 3.0 or later

DRAFT

4. The Unicode Standard

Where some basic level of content is all that is important, phrasing such as in the third example can be used. Where the important information is simply the overall architecture and semantics of the Unicode Standard, the version can be omitted entirely, as in example four.

3.2 Conformance Requirements

This section presents the clauses specifying the formal conformance requirements for processes implementing Version 5.0 of the Unicode Standard. A few of these clauses have been revised from Version 4.0 of the Unicode Standard. The revisions do not change the fundamental substance of the conformance requirements previously set forth, but rather are reformulated to clarify their applicability to Unicode algorithms and tailoring. The definitions that these clauses—particularly C7—depend on have been extended to cover additional aspects of properties and algorithms.

In addition to the specifications printed in this book, the Unicode Standard, Version 5.0, includes a number of Unicode Standard Annexes (UAXes) and the Unicode Character Database. Both are available only electronically, either on the CD-ROM or on the Unicode Web site. At the end of this section there is a list of those annexes that are considered an integral part of the Unicode Standard, Version 5.0.0, and therefore covered by these conformance requirements.

The Unicode Character Database contains an extensive specification of normative and informative character properties completing the formal definition of the Unicode Standard. See *Chapter 4, Character Properties*, for more information.

Note that not all conformance requirements are relevant to all implementations at all times because implementations may not support the particular characters or operations for which a given conformance requirement may be relevant. See *Section 2.14, Conforming to the Unicode Standard*, for more information.

In this section, conformance clauses are identified with the letter *C*.

The numbering of clauses matches that of prior versions of *The Unicode Standard* where possible. Where new clauses were added, letters are used with numbers—for example, *C12a*.

Byte Ordering

C1 [Superseded by C11]

C2 [Superseded by C11]

Earlier versions of the Unicode Standard specified conformance requirements for "code values" (now known as *code units*) in terms of 16-bit values. These requirements have been superseded by the more detailed specification of the Unicode encoding forms: UTF-8, UTF-16, and UTF-32.

C3 [Superseded by C12b]

Earlier versions of the Unicode Standard specified that in the absence of a higher-level protocol, Unicode data serialized into a sequence of bytes would be interpreted most significant byte first. This requirement has been superseded by the more detailed specification of the various Unicode encoding schemes.

DRAFT

Code Points Unassigned to Abstract Characters

C4 *A process shall not interpret a high-surrogate code point or a low-surrogate code point as an abstract character.*

- The high-surrogate and low-surrogate code points are designated for surrogate code units in the UTF-16 character encoding form. They are unassigned to any abstract character.

C5 *A process shall not interpret a noncharacter code point as an abstract character.*

- The noncharacter code points may be used internally, such as for sentinel values or delimiters, but should not be exchanged publicly.

C6 *A process shall not interpret an unassigned code point as an abstract character.*

- This clause does not preclude the assignment of certain generic semantics to unassigned code points (for example, rendering with a glyph to indicate the position within a character block) that allow for graceful behavior in the presence of code points that are outside a supported subset.
- Note that unassigned code points may have default property values. (See D11.)
- Code points whose use has not yet been designated may be assigned to abstract characters in future versions of the standard. Because of this fact, due care in the handling of generic semantics for such code points is likely to provide better robustness for implementations that may encounter data based on future versions of the standard.

Interpretation

C7 *A process shall interpret a coded character sequence according to the character semantics established by this standard, if that process does interpret that coded character sequence.*

- This restriction does not preclude internal transformations that are never visible external to the process.

C8 *A process shall not assume that it is required to interpret any particular coded character sequence.*

- Processes that interpret only a subset of Unicode characters are allowed; there is no blanket requirement to interpret *all* Unicode characters.
- Any means for specifying a subset of characters that a process can interpret is outside the scope of this standard.
- The semantics of a private-use code point is outside the scope of this standard.
- Although these clauses are not intended to preclude enumerations or specifications of the characters that a process or system is able to interpret, they do separate supported subset enumerations from the question of conformance. In actuality, any system may occasionally receive an unfamiliar character code that it is unable to interpret.

C9 *A process shall not assume that the interpretations of two canonical-equivalent character sequences are distinct.*

- The implications of this conformance clause are twofold. First, a process is never required to give different interpretations to two different, but canonical-equivalent character sequences. Second, no process can assume that another process will make a distinction between two different, but canonical-equivalent character sequences.

DRAFT

- Ideally, an implementation would always interpret two canonical-equivalent character sequences identically. There are practical circumstances under which implementations may reasonably distinguish them.
- Even processes that normally do not distinguish between canonical-equivalent character sequences can have reasonable exception behavior. Some examples of this behavior include graceful fallback processing by processes unable to support correct positioning of nonspacing marks; "Show Hidden Text" modes that reveal memory representation structure; and the choice of ignoring collating behavior of combining sequences that are not part of the repertoire of a specified language (see *Section 5.12, Strategies for Handling Nonspacing Marks*).

Modification

C10 *When a process purports not to modify the interpretation of a valid coded character sequence, it shall make no change to that coded character sequence other than the possible replacement of character sequences by their canonical-equivalent sequences or the deletion of* noncharacter *code points.*

- Replacement of a character sequence by a compatibility-equivalent sequence *does* modify the interpretation of the text.
- Replacement or deletion of a character sequence that the process cannot or does not interpret *does* modify the interpretation of the text.
- Changing the bit or byte ordering of a character sequence when transforming it between different machine architectures does not modify the interpretation of the text.
- Changing a valid coded character sequence from one Unicode character encoding form to another does not modify the interpretation of the text.
- Changing the byte serialization of a code unit sequence from one Unicode character encoding scheme to another does not modify the interpretation of the text.
- If a noncharacter that does not have a specific internal use is unexpectedly encountered in processing, an implementation may signal an error or delete or ignore the noncharacter. If these options are not taken, the noncharacter should be treated as an unassigned code point. For example, an API that returned a character property value for a noncharacter would return the same value as the default value for an unassigned code point.
- All processes and higher-level protocols are required to abide by C10 as a minimum. However, higher-level protocols may define additional equivalences that do not constitute modifications under that protocol. For example, a higher-level protocol may allow a sequence of spaces to be replaced by a single space.

Character Encoding Forms

C11 *When a process interprets a code unit sequence which purports to be in a Unicode character encoding form, it shall interpret that code unit sequence according to the corresponding code point sequence.*

- The specification of the code unit sequences for UTF-8 is given in D36.
- The specification of the code unit sequences for UTF-16 is given in D35.
- The specification of the code unit sequences for UTF-32 is given in D31.

DRAFT

C12 *When a process generates a code unit sequence which purports to be in a Unicode character encoding form, it shall not emit ill-formed code unit sequences.*

- The definition of each Unicode character encoding form specifies the ill-formed code unit sequences in the character encoding form. For example, the definition of UTF-8 (D36) specifies that code unit sequences such as <C0 AF> are ill-formed.

C12a *When a process interprets a code unit sequence which purports to be in a Unicode character encoding form, it shall treat ill-formed code unit sequences as an error condition, and shall not interpret such sequences as characters.*

- For example, in UTF-8 every code unit of the form $110xxxx_2$ *must* be followed by a code unit of the form $10xxxxxx_2$. A sequence such as $110xxxxx_2$ $0xxxxxxx_2$ is ill-formed and must never be generated. When faced with this ill-formed code unit sequence while transforming or interpreting text, a conformant process must treat the first code unit $110xxxxx_2$ as an illegally terminated code unit sequence—for example, by signaling an error, filtering the code unit out, or representing the code unit with a marker such as U+FFFD REPLACEMENT CHARACTER.
- Conformant processes cannot interpret ill-formed code unit sequences. However, the conformance clauses do not prevent processes from operating on code unit sequences that do not purport to be in a Unicode character encoding form. For example, for performance reasons a low-level string operation may simply operate directly on code units, without interpreting them as characters. See, especially, the discussion under definition D30e.
- Utility programs are not prevented from operating on "mangled" text. For example, a UTF-8 file could have had CRLF sequences introduced at every 80 bytes by a bad mailer program. This could result in some UTF-8 byte sequences being interrupted by CRLFs, producing illegal byte sequences. This mangled text is no longer UTF-8. It is permissible for a conformant program to repair such text, recognizing that the mangled text was originally well-formed UTF-8 byte sequences. However, such repair of mangled data is a special case, and it must not be used in circumstances where it would cause security problems.

Character Encoding Schemes

C12b *When a process interprets a byte sequence which purports to be in a Unicode character encoding scheme, it shall interpret that byte sequence according to the byte order and specifications for the use of the byte order mark established by this standard for that character encoding scheme.*

- 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.
- For example, when using UTF-16LE, pairs of bytes must be interpreted as UTF-16 code units using the little-endian byte order convention, and any initial <FF FE> sequence is interpreted as U+FEFF ZERO WIDTH NO-BREAK SPACE (part of the text), rather than as a byte order mark (not part of the text). (See D41.)

Bidirectional Text

C13 *A process that displays text containing supported right-to-left characters or embedding codes shall display all visible representations of characters (excluding format characters) in the same order as if the Bidirectional Algorithm had been applied to the text, unless tailored by a higher-level protocol as permitted by the specification.*

DRAFT

- The Bidirectional Algorithm is specified in Unicode Standard Annex #9, "The Bidirectional Algorithm."

[Named algorithms need titlecase. Need global pass. Have done Bidi. 9-29-05]

Normalization Forms

C14 *A process that produces Unicode text that purports to be in a Normalization Form shall do so in accordance with the specifications in Unicode Standard Annex #15, "Unicode Normalization Forms."*

C15 *A process that tests Unicode text to determine whether it is in a Normalization Form shall do so in accordance with the specifications in Unicode Standard Annex #15, "Unicode Normalization Forms."*

C16 *A process that purports to transform text into a Normalization Form must be able to produce the results of the conformance test specified in Unicode Standard Annex #15, "Unicode Normalization Forms."*

- This means that when a process uses the input specified in the conformance test, its output must match the expected output of the test.

Normative References

C17 *Normative references to the Standard itself, to property aliases, to property value aliases, or to Unicode algorithms shall follow the formats specified in Section 3.1, Versions of the Unicode Standard.*

C18 *Higher-level protocols shall not make normative references to provisional properties.*

- Higher-level protocols may make normative references to informative properties.

Unicode Algorithms

C19 *If a process purports to implement a Unicode algorithm, it shall conform to the specification of that algorithm in the standard, including any tailoring by a higher-level protocol as permitted by the specification.*

- The term *Unicode algorithm* is defined at D8a.
- An implementation claiming conformance to a Unicode algorithm need only guarantee that it produces the same results as those specified in the logical description of the process; it is not required to follow the actual described procedure in detail. This allows room for alternative strategies and optimizations in implementation.

C19a *The specification of an algorithm may prohibit or limit tailoring by a higher-level protocol. If a process that purports to implement a Unicode algorithm applies a tailoring, that fact must be disclosed.*

- For example, the algorithms for normalization and canonical ordering are not tailorable. The Bidirectional Algorithm allows some tailoring by higher-level protocols. The Unicode Default Casing algorithms may be tailored without limitation.

Default Casing Algorithms

C20 *An implementation that purports to support Default Case Conversion, Default Case Detection, or Default Caseless Matching shall do so in accordance with the definitions and specifications in Section 3.13, Default Case Algorithms.*

DRAFT

- A conformant implementation may perform casing operations that are different from the default algorithms, perhaps tailored to a particular orthography, so long as the fact that a tailoring is applied is disclosed.

Unicode Standard Annexes

The following standard annexes are approved and considered part of Version 5.0 of the Unicode Standard. These annexes may contain either normative or informative material, or both. Any reference to Version 5.0 of the standard automatically includes these standard annexes.

- UAX #9: The Bidirectional Algorithm, Version 5.0.0
- UAX #11: East Asian Width, Version 5.0.0
- UAX #14: Line Breaking Properties, Version 5.0.0
- UAX #15: Unicode Normalization Forms, Version 5.0.0
- UAX #24: Script Names, Version 5.0.0
- UAX #29: Text Boundaries, Version 5.0.0
- UAX #31: Identifier and Pattern Syntax, Version 5.0.0
- UAX #34: Unicode Named Character Sequences, Version 5.0.0

Conformance to the Unicode Standard requires conformance to the specifications contained in these annexes, as detailed in the conformance clauses listed earlier in this section.

3.3 Semantics

Definitions

This and the following sections more precisely define the terms that are used in the conformance clauses.

The numbering of definitions matches that of prior versions of *The Unicode Standard* where possible. Where new definitions were added, letters are used with numbers—for example, *D7a*. In a few cases, numbers have been reused for definitions.

Character Identity and Semantics

D1 *Normative behavior:* The normative behaviors of the Unicode Standard consist of the following list or any other behaviors specified in the conformance clauses:

5. Character combination
6. Canonical decomposition
7. Compatibility decomposition
8. Canonical ordering behavior
9. Bidirectional behavior, as specified in the Unicode Bidirectional Algorithm (see Unicode Standard Annex #9, "The Bidirectional Algorithm")
10. Conjoining jamo behavior, as specified in *Section 3.12, Conjoining Jamo Behavior*
11. Variation selection, as specified in *Section 16.4, Variation Selectors*

DRAFT

12. Normalization, as specified in Unicode Standard Annex #15, “Unicode Normalization Forms”
13. Default casing, as specified in *Section 3.13, Default Case Algorithms*

D2 [Incorporated into other definitions]

D2a *Character identity:* The identity of a character is established by its character name and representative glyph in *Chapter 17, Code Charts*.

- A character may have a broader range of use than the most literal interpretation of its name might indicate; the coded representation, name, and representative glyph need to be taken in context when establishing the identity of a character. For example, U+002E FULL STOP can represent a sentence period, an abbreviation period, a decimal number separator in English, a thousands number separator in German, and so on. The character name itself is unique, but may be misleading. See “Character Names” in *Section 17.1, Character Names List*.
- Consistency with the representative glyph does not require that the images be identical or even graphically similar; rather, it means that both images are generally recognized to be representations of the same character. Representing the character U+0061 LATIN SMALL LETTER A by the glyph “X” would violate its character identity.

D2b *Character semantics:* The semantics of a character are determined by its identity, normative properties, and behavior.

- Some normative behavior is default behavior; this behavior can be overridden by higher-level protocols. However, in the absence of such protocols, the behavior must be observed so as to follow the character semantics.
- The character combination properties and the canonical ordering behavior cannot be overridden by higher-level protocols. The purpose of this constraint is to guarantee that the order of combining marks in text and the results of normalization are predictable.

3.4 Characters and Encoding

D3 *Abstract character:* A unit of information used for the organization, control, or representation of textual data.

- When representing data, the nature of that data is generally symbolic as opposed to some other kind of data (for example, aural or visual). Examples of such symbolic data include letters, ideographs, digits, punctuation, technical symbols, and dingbats.
- An abstract character has no concrete form and should not be confused with a *glyph*.
- An abstract character does not necessarily correspond to what a user thinks of as a “character” and should not be confused with a *grapheme*.
- The abstract characters encoded by the Unicode Standard are known as Unicode abstract characters.
- Abstract characters not directly encoded by the Unicode Standard can often be represented by the use of combining character sequences.

D4 *Abstract character sequence:* An ordered sequence of abstract characters.

D4a *Unicode codespace:* A range of integers from 0 to $10FFFF_{16}$.

DRAFT

- This particular range is defined for the codespace in the Unicode Standard. Other character encoding standards may use other codespaces.

D4b *Code point:* Any value in the Unicode codespace.

- A code point is also known as a *code position*.
- See D28a for the definition of *code unit*.

D5 *Encoded character:* An association (or mapping) between an abstract character and a code point.

- An encoded character is also referred to as a *coded character*.
- While an encoded character is formally defined in terms of the mapping between an abstract character and a code point, informally it can be thought of as an abstract character taken together with its assigned code point.
- Occasionally, for compatibility with other standards, a single abstract character may correspond to more than one code point—for example, "Å" corresponds both to U+00C5 Å LATIN CAPITAL LETTER A WITH RING ABOVE and to U+212B Å ANGSTROM SIGN.
- A single abstract character may also be *represented* by a sequence of code points—for example, *latin capital letter g with acute* may be represented by the sequence <U+0047 LATIN CAPITAL LETTER G, U+0301 COMBINING ACUTE ACCENT>, rather than being mapped to a single code point.

D6 *Coded character sequence:* A sequence of code points. Normally, this consists of a sequence of encoded characters, but it may also include noncharacters or reserved code points.

- A coded character sequence is also known as a *coded character representation*.
- Internally, a process may choose to make use of noncharacter code points in its coded character sequences. However, such noncharacter code points may not be interpreted as abstract characters (see C5), and their removal by a conformant process does not constitute modification of interpretation of the coded character sequence (see C10).
- Reserved code points are included in coded character sequences, so that the conformance requirements regarding interpretation and modification are properly defined when a Unicode-conformant implementation encounters coded character sequences produced under a future version of the standard.

Unless specified otherwise for clarity, in the text of the Unicode Standard the term *character* alone designates an encoded character. Similarly, the term *character sequence* alone designates a coded character sequence.

D7 [Incorporated into other definitions]

D7a *Deprecated character:* A coded character whose use is strongly discouraged. Such characters are retained in the standard, but should not be used.

- Deprecated characters are retained in the standard so that previously conforming data stay conformant in future versions of the standard. Deprecated characters should not be confused with obsolete characters, which are historical. Obsolete characters do not occur in modern text, but they are not deprecated; their use is not discouraged.

DRAFT

D7b *Noncharacter:* A code point that is permanently reserved for internal use, and that should never be interchanged. Noncharacters consist of the values U+*n*FFFE and U+*n*FFFF (where *n* is from 0 to 10_{16}) and the values U+FDD0..U+FDEF.

- For more information, see *Section 16.7, Noncharacters*.
- These code points are permanently reserved as noncharacters.

D7c *Reserved code point:* Any code point of the Unicode Standard which is reserved for future assignment. Also known as an *unassigned code point*.

- Note that surrogate code points and noncharacters are considered assigned code points, but not assigned characters.
- For a summary classification of reserved and other types of code points, see *Table 2-3*.

In general, a conforming process may indicate the presence of a code point whose use has not been designated (for example, by showing a missing glyph in rendering, or by signaling an appropriate error in a streaming protocol), even though it is forbidden by the standard from *interpreting* that code point as an abstract character.

D8 *Higher-level protocol:* Any agreement on the interpretation of Unicode characters that extends beyond the scope of this standard.

- Such an agreement need not be formally announced in data; it may be implicit in the context.
- The specification of some Unicode algorithms may limit the scope of what a conformant higher-level protocol may do.

D8a *Unicode algorithm:* The logical description of a process used to achieve a specified result involving Unicode characters.

- This definition, used in the Unicode Standard and other publications of the Unicode Consortium, is intentionally broad, so as to allow precise logical description of required results, without constraining implementations to follow the precise steps of that logical description.

D8b *Named Unicode algorithm:* A Unicode algorithm which is specified in the Unicode Standard or in other standards published by the Unicode Consortium and which is given an explicit name for ease of reference.

- Named Unicode algorithms are cited in titlecase in the Standard.
- When referenced outside the context of the Unicode Standard, it is customary to prepend the word “Unicode” to the name of the algorithm.

Table 3-1 lists the named Unicode algorithms and indicates the location of their specification. Details regarding conformance to these algorithms and any restrictions they place on the scope of allowable tailoring by higher-level protocols can be found in the specifications. In some cases, a named Unicode algorithm is provided for information only.

Table 3-1. Named Unicode Algorithms

Name	Description
Canonical Ordering	*Section 3.11*
Hangul Syllable Boundary Determination	*Section 3.12*
Hangul Syllable Composition	*Section 3.12*
Hangul Syllable Decomposition	*Section 3.12*
Hangul Syllable Name Generation	*Section 3.12*

DRAFT

Table 3-1. Named Unicode Algorithms (Continued)

Name	Description
Default Case Conversion	*Section 3.13*
Default Case Detection	*Section 3.13*
Default Caseless Matching	*Section 3.13*, *Section 5.18*
Bidirectional Algorithm	UAX #9
Line Breaking Algorithm	UAX #14
Normalization Algorithm	UAX #15
Grapheme Cluster Boundary Determination	UAX #29
Word Boundary Determination	UAX #29
Sentence Boundary Determination	UAX #29
Default Identifier Determination	UAX #31
Alternative Identifier Determination	UAX #31
Pattern Syntax Determination	UAX #31
Identifier Normalization	UAX #31
Identifier Case Folding	UAX #31
Standard Compression Scheme for Unicode (SCSU)	UTS #6
Collation Algorithm (UCA)	UTS #10

3.5 Properties

The Unicode Standard specifies many different types of character properties. This section provides the basic definitions related to character properties.

The actual values of Unicode character properties are specified in the Unicode Character Database. See *Section 4.1, Unicode Character Database*, for an overview of those data files. *Chapter 4, Character Properties*, contains more detailed descriptions of some particular, important character properties. Additional properties that are specific to particular characters (such as the definition and use of the *right-to-left override* character or *zero-width space*) are discussed in the relevant sections of this standard.

The interpretation of some properties (such as the case of a character) is independent of context, whereas the interpretation of other properties (such as directionality) is applicable to a character sequence as a whole, rather than to the individual characters that compose the sequence.

[The K definitions (K is a temporary random choice to put them in FM in a similiar style without affecting the numbering of the current definitions) and some related material are new, partly from UTR #23. JDA 10-1-05]

Types of Properties

K1 *Property:* A named attribute of an entity in the Unicode Standard, associated with a defined set of values.

K2 *Code point property:* A property of code points.

- Code point properties refer to attributes of code points per se, based on architectural considerations of this standard, irrespective of any particular encoded character.
- Thus, the Surrogate property and the Noncharacter property are code point properties.

K3 *Abstract character property:* A property of abstract characters.

DRAFT

- Abstract character properties refer to attributes of abstract characters per se, based on their independent existence as elements of writing systems or other notational systems, irrespective of their encoding in the Unicode Standard.
- Thus, the Alphabetic property, the Punctuation property, the Hex_Digit property, the Numeric_Value property, and so on, are properties of abstract characters and are associated with those characters whether encoded in the Unicode Standard or in any other character encoding—or even prior to their being encoded in any character encoding standard.

K4 *Encoded character property:* A property of encoded characters in the Unicode Standard.

- For each encoded character property there is a mapping from every code point to some value in the set of values associated with that property.

Encoded character properties are defined this way to facilitate the implementation of character property APIs based on the Unicode Character Database. Typically, an API will take a property and a code point as input, and will return a value for that property as output, interpreting it as the "character property" for the "character" encoded at that code point. However, to be useful, such APIs must return meaningful values for unassigned code points, as well as for encoded characters.

In some instances an encoded character property in the Unicode Standard is exactly equivalent to a code point property. For example, the Pattern_Syntax property simply defines a range of code points which are reserved for pattern syntax. (See Unicode Standard Annex #31, "Identifier and Pattern Syntax.")

In other instances, an encoded character property directly reflects an abstract character property, but extends the domain of the property to include all code points, including unassigned code points. For Boolean properties, such as the Hex_Digit property, typically an encoded character property will be true for the encoded characters with that abstract character property, and will be false for all other code points, including unassigned code points, noncharacters, private-use characters, and encoded characters for which the abstract character property is inapplicable or irrelevant.

However, in many instances, an encoded character property is semantically complex, and may telescope together values associated with a number of abstract character properties and/or code point properties. The General_Category property is an example—it contains values associated with several abstract character properties (such as Letter, Punctuation, Symbol), and also code point properties (such as [gc=Cs] for the Surrogate code point property).

In the text of this standard the terms "Unicode character property," "character property," or the term "property" without qualifier, generally refer to an encoded character property, unless otherwise indicated.

A list of the encoded character properties formally considered to be a part of the Unicode Standard can be found in PropertyAliases.txt in the Unicode Character Database. See also "Property Aliases" later in this section.

Property Values

K5 *Property value:* One of the set of values associated with an encoded character property.

- For example, the East_Asian_Width [EAW] property has the possible values "Narrow", "Neutral", "Wide", "Ambiguous", and "Unassigned".

DRAFT

A list of the values associated with encoded character properties in the Unicode Standard can be found in PropertyValueAliases.txt in the Unicode Character Database. See also "Property Aliases" later in this section.

K6 *Explicit property value:* A value for an encoded character property which is explicitly associated with a code point in one of the data files of the Unicode Character Database.

K7 *Implicit property value:* A value for an encoded character property which is given by generic rule or by an "otherwise" clause in one of the data files of the Unicode Character Database.

- Implicit property values are used to avoid having to explicitly list values for over a million code points (most of them unassigned) for every property.

K8 *Default property value:* The value (or in some cases small set of values) of a property associated with unassigned code points or with encoded characters for which the property is irrelevant.

- For example, for most Boolean properties, "False" is the default property value. Note that in such cases, the default property value used for unassigned code points may be the same value that is used for many assigned characters as well.
- Some properties, particularly enumerated properties, specify a particular, unique value as their default value. For example, "XX" is the default property value for the Line_Break property.
- A default property value is typically defined implicitly, to avoid having to repeat long lists of unassigned code points.
- In the case of some properties with arbitrary string values, the default property value is an implied null value. For example, the fact that there is no Unicode character name for unassigned code points is equivalent to saying that the default property value for the Name property for an unassigned code point is a null string.
- In some instances, an encoded character property may have multiple default values. For example, the Bidi_Class property defines a range of unassigned code points as having the "R" value, another range of unassigned code points the "AL" value, and the otherwise case as having the "L" value.

Classification of Properties by Their Values

K9 *Enumerated property:* A property with a small set of named values.

- As characters are added to the Unicode Standard, the set of values may need to be extended in the future, but enumerated properties have a relatively fixed set of possible values.

K10 *Closed enumeration:* An enumerated property for which the set of values is closed and will not be extended for future versions of the Unicode Standard.

- Currently, the General Category is the only closed enumeration, except for the Boolean properties.

K11 *Boolean property:* A closed enumerated property whose set of values is limited to "true" and "false".

- The presence or absence of the property is the essential information.

K12 *Numeric property:* A numeric property is a property whose value is a number that can take on any integer, or real value.

DRAFT

- An example is the Numeric_Value property. There is no implied limit to the number of possible distinct values for the property, except the limitations on representing integers or real numbers in computers.

K13 *String-valued property:* A property whose value is a string.

- The Canonical_Decomposition property is a string-valued property.

K14 *Catalog property:* A property that is an enumerated property, typically unrelated to an algorithm, that may be extended in each successive version of the Unicode Standard.

- Examples are Age and Block properties. Additional values for both may be added each time a new version of the Standard adds new characters or blocks.

K15 *Miscellaneous property:* A property whose values are not Boolean, enumerated, numeric, string or catalog values.

- The Unicode character name property is a miscellaneous property.

Normative and Informative Properties

Unicode character properties are divided into those that are normative and those that are informative.

D9 *Normative property:* A Unicode character property used in the specification of the standard.

Specification that a character property is *normative* means that implementations which claim conformance to a particular version of the Unicode Standard and which make use of that particular property must follow the specifications of the standard for that property for the implementation to be conformant. For example, the directionality property (bidirectional character type) is required for conformance whenever rendering text that requires bidirectional layout, such as Arabic or Hebrew.

Whenever a normative process depends on a property in a specified way, that property is designated as normative.

The fact that a given Unicode character property is normative does *not* mean that the values of the property will never change for particular characters. Corrections and extensions to the standard in the future may require minor changes to normative values, even though the Unicode Technical Committee strives to minimize such changes. See also "Stability of Properties" later in this section.

Some of the normative Unicode algorithms depend critically on particular property values for their behavior. Normalization, for example, defines an aspect of textual interoperability that many applications rely on to be absolutely stable. As a result, some of the normative properties disallow any kind of overriding by higher-level protocols. Thus, the decomposition of Unicode characters is both normative and *not overridable*; no higher-level protocol may override these values, because to do so would result in non-interoperable results for the normalization of Unicode text. Other normative properties, such as case mapping, are *overridable* by higher-level protocols, because their intent is to provide a common basis for behavior, but they may require tailoring for particular local cultural conventions or particular implementations.

Some important normative character properties of the Unicode Standard are listed in *Table 3-2*, with an indication of which sections in the standard provide a general description of the properties and their use. Other normative properties are documented in the Unicode Character Database. In all cases, the Unicode Character Database provides the definitive list of character properties and the exact list of property value assignments for

DRAFT

each version of the Standard. A list of additional, special character properties can be found in *Section 4.12, Characters with Unusual Properties*.

Table 3-2. Normative Character Properties

Property	Description
Bidi_Class (directionality)	UAX #9 and *Section 4.4*
Bidi_Mirrored	*Section 4.7* and UAX #9
Block	*Chapter 17*
Canonical_Combining_Class	*Section 3.11*, *Section 4.3*, and UAX #15
Case-related properties	*Section 3.13*, *Section 4.2*, and *Chapter 17*
Composition_Exclusion	UAX #15
Decomposition_Mapping	*Chapter 3*, *Chapter 17*, and UAX #15
Default_Ignorable_Code_Point	*Section 5.20*
Deprecated	*Section 3.1*
General_Category	*Section 4.5*
Hangul_Syllable_Type	*Section 3.12* and UAX #29
Jamo_Short_Name	*Section 3.12*
Joining_Type and Joining_Group	*Section 8.2*
Name	*Chapter 17*
Noncharacter_Code_Point	*Section 16.7*
Numeric_Value	*Section 4.6*
White_Space	UCD.html

K16 *Overridable property:* A normative property whose values may be overridden by conformant higher-level protocols.

- For example, the Canonical_Decomposition property is not overridable. The Uppercase property can be overridden.

D9a *Informative property:* A Unicode character property whose values are provided for information only.

A conformant implementation of the Unicode Standard is free to use or change informative property values as it may require, while remaining conformant to the standard. An implementer always has the option of establishing a protocol to convey the fact that informative properties are being used in distinct ways.

Informative properties capture expert implementation experience. When an informative property is explicitly specified in the Unicode Character Database, its use is strongly recommended for implementations to encourage comparable behavior between implementations. Note that it is possible for an informative property in one version of the Unicode Standard to become a normative property in a subsequent version of the standard if its use starts to acquire conformance implications in some part of the standard.

Table 3-3 provides a partial list of the more important informative character properties. For a complete listing, see the Unicode Character Database.

Table 3-3. Informative Character Properties

Property	Description
Dash	*Section 6.2* and *Table 6-3*
East_Asian_Width	*Section 12.3* and UAX #11
Letters-related properties	*Section 4.10*
Line_Break	*Section 16.1*, *Section 16.2*, and UAX #14
Mathematical	*Section 15.4*

DRAFT

Table 3-3. Informative Character Properties (Continued)

Property	**Description**
Script	UAX #24
Space	*Section 6.2* and *Table 6-2*
Unicode_1_Name	*Section 4.9*

D9b *Provisional property:* A Unicode character property whose values are unapproved and tentative, and which may be incomplete or otherwise not in a usable state.

- Provisional properties may be removed from future versions of the standard, without prior notice.

Some of the information provided about characters in the Unicode Character Database constitutes provisional data. This may capture partial or preliminary information. It may contain errors or omissions, or otherwise not be ready for systematic use; however, it is included in the data files for distribution partly to encourage review and improvement of the information. For example, a number of the tags in the Unihan.txt file provide provisional property values of various sorts about Han characters.

The data files of the Unicode Character Database may also contain various annotations and comments about characters, and those annotations and comments should be considered provisional. Implementations should not attempt to parse annotations and comments out of the data files and treat them as informative character properties per se.

Context Dependence

K17 *Context-dependent property:* A property that applies to a code point in the context of a longer code point sequence.

- For example, the lowercase mapping of a Greek sigma depends on the context of the surrounding characters.

K18 *Context-independent property:* A property that is not context dependent: it applies to a code point in isolation.

Stability of Properties

K19 *Stable transformation:* A transformation T on a property P is stable with respect to an algorithm A, if the result of the algorithm on the transformed property $A(T(P))$ is the same as the original result $A(P)$ for all code points.

K20 *Stable property:* A property is stable with respect to a particular algorithm or process, as long as possible changes in the assignment of property values are restricted in such a manner that the result of the algorithm on the property continues to be the same as the original result for all previously assigned code points.

- For example, while the absolute values of the canonical combining classes are *not* guaranteed to be the same between versions of the Unicode Standard, their relative values will be maintained. As a result, the Canonical Combining Class, while not immutable, is a stable property with respect to the Normalization Forms as defined in Unicode Standard Annex #15, “Unicode Normalization Forms.”
- As new characters are assigned to previously unassigned code points, replacing any default values for these code points with actual property values must maintain stability.

DRAFT

K21 *Fixed property:* A property whose values (other than a default value), once associated with a specific code point, are fixed and will not be changed, except to correct obvious or clerical errors.

- For a fixed property, any default values can be replaced without restriction by actual property values, as new characters are assigned to previously unassigned code points. Examples of fixed properties are Age or Hangul_Syllable_Type.
- Designating a property as fixed does not imply stability or immutability, see "Stability" in *Section 3.1, Versions of the Unicode Standard*. While the age of a character, for example, is established by the version of the Unicode Standard at which it was added, errors in the published listing of the property value could be corrected. For some other properties, there are explicit stability guarantees that prohibit the correction even of such errors.

K22 *Immutable property:* A fixed property that is also subject to a stability guarantee preventing *any* change in the published listing of property values other than assignment of new values to formerly unassigned code points.

- An immutable property is trivially stable with respect to *all* context-free algorithms.

[Mark says we haven't defined "context-free algorithms"--needs at least a couple of examples (+ and -). JDA 10-1-05]

- An example of an immutable property is the Unicode character name itself. Because character names are values of an immutable property, misspellings and incorrect names will *never* be corrected clerically. Any errata will be noted in a comment in the character names list, and, where needed, an informative character name alias will be provided.
- When an encoded character property representing a code point property is immutable, none of its values can ever change. This follows from the fact that the code points themselves do not change, and the status of the property is unaffected by whether a particular abstract character is encoded at a code point later or not. An example of such a property is the Pattern_Syntax property; all values of that property are unchangeable for all code points, forever.
- In the more typical case of an immutable property, the values for existing encoded characters cannot change, but when a new character is encoded, the formerly unassigned code point changes from having a default value for the property to having one of its non-default values. Once that non-default value is published, then it can no longer be changed.

K23 *Stabilized property:* A property which is neither extended to new characters, nor maintained in any other manner, but which is retained in the Unicode Character Database.

- A stabilized property is also a fixed property.

K24 *Deprecated property:* A property whose use by implementations is discouraged.

- One of the reasons a property may be deprecated is because a different combination of properties better expresses the intended semantics.
- Where sufficiently widespread legacy support exists for the deprecated property, not all implementations may be able to discontinue the use of the deprecated property. In such a case, a deprecated property may be extended to new characters, so as to maintain it in a usable and consistent state.

DRAFT

Informative or normative properties in the standard will not be removed even when they are supplanted by other properties or are no longer useful. However, they may be stabilized and/or deprecated.

Simple and Derived Properties

D9c *Simple property:* A Unicode character property whose values are specified directly in the Unicode Character Database (or elsewhere in the standard) and whose values cannot be derived from other simple properties.

D9d *Derived property:* A Unicode character property whose values are algorithmically derived from some combination of simple properties.

[4.0.1 D definition updates need to be put in glossary also. Figure out which they are and do. 8-24-04.]

The Unicode Character Database lists a number of derived properties explicitly. Even though these values can be derived, they are provided as lists because the derivation may not be trivial and because explicit lists are easier to understand, reference, and implement. Good examples of derived properties are the ID_Start and ID_Continue properties, which can be used to specify a formal identifier syntax for Unicode characters. The details of how derived properties are computed can be found in the documentation for the Unicode Character Database.

Property Aliases

To enable normative references to Unicode character properties, formal aliases for properties and for property values are defined as part of the Unicode Character Database.

D10 *Property alias:* A unique identifier for a particular Unicode character property.

- The identifiers used for property aliases contain only ASCII alphanumeric characters or the underscore character.
- Short and long forms for each property alias are defined. The short forms are typically just two or three characters long to facilitate their use as attributes for tags in markup languages. For example, "General_Category" is the long form and "gc" is the short form of the property alias for the General Category property.
- Property aliases are defined in the file PropertyAliases.txt in the Unicode Character Database.
- Property aliases of normative properties are themselves normative.

D10a *Property value alias:* A unique identifier for a particular enumerated value for a particular Unicode character property.

- The identifiers used for property value aliases contain only ASCII alphanumeric characters or the underscore character, or have the special value "n/a".
- Short and long forms for property value aliases are defined. For example, "Currency_Symbol" is the long form and "Sc" is the short form of the property value alias for the currency symbol value of the General Category property.
- Property value aliases are defined in the file PropertyValueAliases.txt in the Unicode Character Database.
- Property value aliases are unique identifiers only in the context of the particular property with which they are associated. The same identifier string might be associ-

DRAFT

ated with an entirely different value for a different property. The combination of a property alias and a property value alias is, however, guaranteed to be unique.

- Property value aliases referring to values of normative properties are themselves normative.

The property aliases and property value aliases can be used, for example, in XML formats of property data, for regular-expression property tests, and in other programmatic textual descriptions of Unicode property data. Thus "gc=Lu" is a formal way of specifying that the General Category of a character (using the property alias "gc") has the value of being an uppercase letter (using the property value alias "Lu").

D11 [Revised to Definition K8]

Private Use

D12 *Private-use code point:* Code points in the ranges U+E000..U+F8FF, U+F0000.. U+FFFFD, and U+100000..U+10FFFD.

- Private-use code points are considered to be assigned characters, but the abstract characters associated with them have no interpretation specified by this standard. They can be given any interpretation by conformant processes.
- Private-use code points may be given default property values, but these default values are overridable by higher-level protocols that give those private-use code points a specific interpretation.

3.6 Combination

D13 [Revised to Definition D13b]

D13a *Graphic character:* A character with the General Categories of Letter (L), Combining Mark (M), Number (N), Punctuation (P), Symbol (S), or Space Separator (Zs).

- Graphic characters specifically exclude the line and paragraph separators (Zl, Zp), and exclude the characters with the General Categories of Other (Cn, Cs, Cc, Cf).
- The interpretation of private-use characters (Co) as graphic characters or not is determined by the implementation.
- For more information, see *Chapter 2, General Structure*, especially *Section 2.4, Code Points and Characters*, and *Table 2-3*.

D13b *Base character:* Any graphic character except for those with the General Category of Combining Mark (M).

- Most Unicode characters are base characters. In terms of General Category values, a base character is any code point that has one of the categories: Letter (L), Number (N), Punctuation (P), Symbol (S), or Space Separator (Zs).
- Base characters do not include control characters or format controls.
- Base characters are independent graphic characters, but this does not preclude the presentation of base characters from adopting different contextual forms or participating in ligatures.
- The interpretation of private-use characters (Co) as base characters or not is determined by the implementation. However, the default interpretation of private-use characters should be as base characters, in the absence of other information.

DRAFT

D14 *Combining character:* A character with the General Category of Combining Mark (M).

- Combining characters consist of all characters with the General Category values of Spacing Combining Mark (Mc), Non-Spacing Mark (Mn), and Enclosing Mark (Me).
- All characters with non-zero canonical combining class are combining characters, but the reverse is not the case: there are combining characters with a zero canonical combining class.
- The interpretation of private-use characters (Co) as combining characters or not is determined by the implementation.
- These characters are not normally used in isolation unless they are being described. They include such characters as accents, diacritics, Hebrew points, Arabic vowel signs, and Indic matras.
- The graphic positioning of a combining character depends on the last preceding base character, unless they are separated by a character that is neither a combining character nor one of ZERO WIDTH JOINER or ZERO WIDTH NON-JOINER. The combining character is said to *apply* to that base character.
- There may be no such base character, such as when a combining character is at the start of text or follows a control or format character, such as a carriage return, tab, or RIGHT-LEFT MARK. In such cases, the combining characters are called *isolated combining characters.*
- With isolated combining characters, or when a process is unable to perform graphical combination, a process may present a combining character without graphical combination; that is, it may present it as if it were a base character.
- The representative images of combining characters are depicted with a dotted circle in the code charts; when presented in graphical combination with a preceding base character, that base character is intended to appear in the position occupied by the dotted circle.
- Combining characters generally take on the properties of their base character, while retaining their combining property.

D15 *Nonspacing mark:* A combining character with the property [General_Category = Mn] or [General_Category = Me].

- The position of a nonspacing mark in presentation is dependent on its base character. It generally does not consume space along the visual baseline in and of itself.
- Such characters may be large enough to affect the placement of their base character relative to preceding and succeeding base characters. For example, a circumflex applied to an "i" may affect spacing ("î"), as might the character U+20DD COMBINING ENCLOSING CIRCLE.

D15a *Enclosing mark:* A nonspacing mark with the property [General_Category = Me].

- Enclosing marks are a subclass of nonspacing marks which surround a base character, rather than merely being placed over, under, or through it.

D16 *Spacing mark:* A combining character that is not a nonspacing mark.

- Examples include U+093F DEVANAGARI VOWEL SIGN I. In general, the behavior of spacing marks does not differ greatly from that of base characters.

DRAFT

- Note that spacing marks such as U+0BCA TAMIL VOWEL SIGN O may appear on both sides of a base character, but are not enclosing marks.

D17 *Combining character sequence:* A maximal character sequence consisting of either a base character followed by a sequence of one or more characters where each is a combining character, ZERO WIDTH JOINER, or ZERO WIDTH NON-JOINER; or a sequence of one or more characters where each is a combining character, ZERO WIDTH JOINER, or ZERO WIDTH NON-JOINER.

- When identifying a combining character sequence in Unicode text, the definition of the combining character sequence is applied maximally. Thus, for example, in the sequence <c, dot-below, caron, acute, a>, the entire sequence <c, dot-below, caron, acute> is identified as the combining character sequence, rather than the alternative of identifying <c, dot-below> as a combining character sequence followed by a separate (defective) combining character sequence <caron, acute>.

D17a *Defective combining character sequence:* A combining character sequence that does not start with a base character.

- Defective combining character sequences occur when a sequence of combining characters appears at the start of a string or follows a control or format character. Such sequences are defective from the point of view of handling of combining marks, but are not *ill-formed*. (See D30.)

D17b *Grapheme base:* A character with the property [Grapheme_Base = True], or any standard Korean syllable block.

- Characters with the property [Grapheme_Base = True] include all base characters plus most spacing marks.
- The concept of a grapheme base is introduced to simplify discussion of the graphical application of nonspacing marks to other elements of text. Note that a grapheme base may consist of a spacing (combining) mark, which distinguishes it from a base character, per se. A grapheme base may also itself consist of a sequence of characters, in the case of the standard Korean syllable block.
- For the definition of standard Korean syllable block, see K21 in *Section 3.12, Conjoining Jamo Behavior*.

[Update K21 ref above as necessary once reformatted. JDA 9-27-05]

D17c *Grapheme extender:* A character with the property [Grapheme_Extend = True].

- Grapheme extender characters consist of all nonspacing marks, ZERO WIDTH JOINER, ZERO WIDTH NON-JOINER, and a small number of spacing marks.
- A grapheme extender can be conceived of primarily as the kind of nonspacing graphical mark which gets applied above or below another spacing character.
- ZERO WIDTH JOINER and ZERO WIDTH NON-JOINER are formally defined to be grapheme extenders so that their presence does not break up a sequence of other grapheme extenders.
- The small number of spacing marks which have the property [Grapheme_Extend = True] are all the second parts of a two-part combining mark.

D17d *Grapheme cluster:* A maximal character sequence consisting of a grapheme base followed by zero or more grapheme extenders, or alternatively the sequence <CR, LF>.

- The grapheme cluster represents a horizontally segmentable unit of text, consisting of some grapheme base (which may consist of a Korean syllable) together with any number of nonspacing marks applied to it.

DRAFT

- A grapheme cluster is similar, but not identical to a combining character sequence. A combining character sequence starts with a *base* character, and extends across any subsequent sequence of combining marks, *nonspacing* or *spacing*. A combining character sequence is most directly relevant to processing issues related to normalization, comparison, and searching.
- A grapheme cluster starts with a *grapheme base*, and extends across any subsequent sequence of *nonspacing* marks. A grapheme cluster is most directly relevant to text rendering and such processes as cursor placement and text selection in editing.

D17e *Extended grapheme cluster:* The text between grapheme cluster boundaries as specified by Unicode Standard Annex #29, "Text Boundaries."

- Extended grapheme clusters are either a grapheme cluster, a single character such as a control character, or the sequence <CR, LF>. They do not have linguistic significance, but are used to break up a string of text into units for processing.

3.7 Decomposition

D17f *Decomposition mapping:* A mapping from a character to a sequence of one or more characters that is a canonical or compatibility equivalent, and listed in the character names list or described in *Section 3.12, Conjoining Jamo Behavior*.

- Each character has at most one decomposition mapping. The mappings in *Section 3.12, Conjoining Jamo Behavior* are canonical mappings. The mappings in the character names list are identified as either canonical or compatibility mappings (see *Section 17.1, Character Names List*).

D18 *Decomposable character:* A character that is equivalent to a sequence of one or more other characters, according to the decomposition mappings found in the Unicode Character Database, and those described in *Section 3.12, Conjoining Jamo Behavior*. It may also be known as a *precomposed* character or *composite* character.

- The decomposition mappings from the Unicode Character Database are also printed in *Section 17.1, Character Names List*.

D19 *Decomposition:* A sequence of one or more characters that is equivalent to a decomposable character. A full decomposition of a character sequence results from decomposing each of the characters in the sequence until no characters can be further decomposed.

Compatibility Decomposition

D20 *Compatibility decomposition:* The decomposition of a character that results from recursively applying *both* the compatibility mappings *and* the canonical mappings found in the Unicode Character Database, and those described in *Section 3.12, Conjoining Jamo Behavior*, until no characters can be further decomposed, and then reordering nonspacing marks according to *Section 3.11, Canonical Ordering Behavior*.

- The decomposition mappings from the Unicode Character Database are also printed in *Section 17.1, Character Names List*.
- Some compatibility decompositions remove formatting information.

D21 *Compatibility decomposable character:* A character whose compatibility decomposition is not identical to its canonical decomposition. It may also be known as a *compatibility precomposed* character or a *compatibility composite* character.

- For example, U+00B5 MICRO SIGN has no canonical decomposition mapping, so its canonical decomposition is the same as the character itself. It has a compatibility decomposition to U+03BC GREEK SMALL LETTER MU. Because MICRO SIGN has a compatibility decomposition that is not equal to its canonical decomposition, it is a compatibility decomposable character.
- For example, U+03D3 GREEK UPSILON WITH ACUTE AND HOOK SYMBOL canonically decomposes to the sequence <U+03D2 GREEK UPSILON WITH HOOK SYMBOL, U+0301 COMBINING ACUTE ACCENT>. That sequence has a compatibility decomposition of <U+03A5 GREEK CAPITAL LETTER UPSILON, U+0301 COMBINING ACUTE ACCENT>. Because GREEK UPSILON WITH ACUTE AND HOOK SYMBOL has a compatibility decomposition that is not equal to its canonical decomposition, it is a compatibility decomposable character.
- This should not be confused with the term "compatibility character," which is discussed in *Section 2.3, Compatibility Characters*.
- Compatibility decomposable characters are a subset of compatibility characters included in the Unicode Standard to represent distinctions in other base standards. They support transmission and processing of legacy data. Their use is discouraged other than for legacy data or other special circumstances.
- Replacing a compatibility decomposable character by its compatibility decomposition may lose round-trip convertibility with a base standard.

D22 *Compatibility equivalent:* Two character sequences are said to be compatibility equivalents if their full compatibility decompositions are identical.

Canonical Decomposition

D23 *Canonical decomposition:* The decomposition of a character that results from recursively applying the canonical mappings found in the Unicode Character Database, and those described in *Section 3.12, Conjoining Jamo Behavior*, until no characters can be further decomposed, and then reordering nonspacing marks according to *Section 3.11, Canonical Ordering Behavior*.

- The decomposition mappings from the Unicode Character Database are also printed in *Section 17.1, Character Names List.*
- A canonical decomposition does not remove formatting information.

D23a *Canonical decomposable character:* A character which is not identical to its canonical decomposition. It may also be known as a *canonical precomposed* character or a *canonical composite* character.

- For example, U+00E0 LATIN SMALL LETTER A WITH GRAVE is a canonical decomposable character because its canonical decomposition is to the sequence <U+0061 LATIN SMALL LETTER A, U+0300 COMBINING GRAVE ACCENT>. U+212A KELVIN SIGN is a canonical decomposable character because its canonical decomposition is to U+004B LATIN CAPITAL LETTER K.

D24 *Canonical equivalent:* Two character sequences are said to be canonical equivalents if their full canonical decompositions are identical.

- For example, the sequences <*o, combining-diaeresis*> and <*ö*> are canonical equivalents. Canonical equivalence is a Unicode property. It should not be confused with language-specific collation or matching, which may add other equivalencies. For example, in Swedish, *ö* is treated as a completely different letter from *o*, collated after *z*. In German, *ö* is weakly equivalent to *oe* and collated with *oe*. In English, *ö* is just

DRAFT

an *o* with a diacritic that indicates that it is pronounced separately from the previous letter (as in *coöperate*) and is collated with *o*.

- By definition, all canonical-equivalent sequences are also compatibility-equivalent sequences.

Note: For information on the use of decomposition in normalization, see Unicode Standard Annex #15, "Unicode Normalization Forms."

3.8 Surrogates

D25 *High-surrogate code point:* A Unicode code point in the range U+D800 to U+DBFF.

D25a *High-surrogate code unit:* A 16-bit code unit in the range $D800_{16}$ to $DBFF_{16}$, used in UTF-16 as the leading code unit of a surrogate pair.

D26 *Low-surrogate code point:* A Unicode code point in the range U+DC00 to U+DFFF.

D26a *Low-surrogate code unit:* A 16-bit code unit in the range $DC00_{16}$ to $DFFF_{16}$, used in UTF-16 as the trailing code unit of a surrogate pair.

- High-surrogate and low-surrogate code points are designated only for that use.
- High-surrogate and low-surrogate code units are used *only* in the context of the UTF-16 character encoding form.

D27 *Surrogate pair:* A representation for a single abstract character that consists of a sequence of two 16-bit code units, where the first value of the pair is a high-surrogate code unit, and the second is a low-surrogate code unit.

- Surrogate pairs are used only in UTF-16. (See *Section 3.9, Unicode Encoding Forms.*)
- Isolated surrogate code units have no interpretation on their own. Certain other isolated code units in other encoding forms also have no interpretation on their own. For example, the isolated byte 80_{16} has no interpretation in UTF-8; it can *only* be used as part of a multibyte sequence. (See *Table 3-7.*)
- Sometimes high-surrogate code units are referred to as *leading surrogates*. Low-surrogate code units are then referred to as *trailing surrogates*. This is analogous to usage in UTF-8, which has *leading bytes* and *trailing bytes*.
- For more information, see *Section 16.6, Surrogates Area*, and *Section 5.4, Handling Surrogate Pairs in UTF-16*.

3.9 Unicode Encoding Forms

The Unicode Standard supports three character encoding forms: UTF-32, UTF-16, and UTF-8. Each encoding form maps the Unicode code points U+0000..U+D7FF and U+E000..U+10FFFF to unique code unit sequences. The size of the code unit is specified for each encoding form. This section presents the formal definition of each of these encoding forms.

D28 *Unicode scalar value:* Any Unicode code point except high-surrogate and low-surrogate code points.

- As a result of this definition, the set of Unicode scalar values consists of the ranges 0 to $D7FF_{16}$ and $E000_{16}$ to $10FFFF_{16}$, inclusive.

DRAFT

D28a *Code unit:* The minimal bit combination that can represent a unit of encoded text for processing or interchange.

- Code units are particular units of computer storage. Other character encoding standards typically use code units defined as 8-bit units, or *octets*. The Unicode Standard uses 8-bit code units in the UTF-8 encoding form, 16-bit code units in the UTF-16 encoding form, and 32-bit code units in the UTF-32 encoding form.
- A code unit is also referred to as a *code value* in the information industry.
- In the Unicode Standard, specific values of some code units cannot be used to represent an encoded character in isolation. This restriction applies to isolated surrogate code units in UTF-16 and to the bytes 80–FF in UTF-8. Similar restrictions apply for the implementations of other character encoding standards; for example, the bytes 81–9F, E0–FC in SJIS (Shift-JIS) cannot represent an encoded character by themselves.
- For information on use of `wchar_t` or other programming language types to represent Unicode code units, see *Section 5.2, ANSI/ISO C wchar_t*.

D28b *Code unit sequence:* An ordered sequence of one or more code units.

- When the code unit is an 8-bit unit, a code unit sequence may also be referred to as a *byte sequence*.
- Note that a code unit sequence may consist of a single code unit.
- In the context of programming languages, the *value* of a *string* data type basically consists of a code unit sequence. Informally, a code unit sequence is itself just referred to as a *string*, and a *byte sequence* is referred to as a *byte string*. Care must be taken in making this terminological equivalence, however, because the formally defined concept of string may have additional requirements or complications in programming languages. For example, a *string* is defined as a *pointer to char* in the C language, and is conventionally terminated with a NULL character. In object-oriented languages, a *string* is a complex object, with associated methods, and its value may or may not consist of merely a code unit sequence.
- Depending on the structure of a character encoding standard, it may be necessary to use a code unit sequence (of more than one unit) to represent a single encoded character. For example, the code unit in SJIS is a byte: Encoded characters such as "a" can be represented with a single byte in SJIS, whereas ideographs require a sequence of two code units. The Unicode Standard also makes use of code unit sequences whose length is greater than one code unit.

D29 A *Unicode encoding form* assigns each Unicode scalar value to a unique code unit sequence.

- For historical reasons, the Unicode encoding forms are also referred to as *Unicode* (or *UCS*) *transformation formats* (UTF). That term is, however, ambiguous between its usage for encoding forms and encoding schemes.
- The mapping of the set of Unicode scalar values to the set of code unit sequences for a Unicode encoding form is *one-to-one*. This property guarantees that a reverse mapping can always be derived. Given the mapping of any Unicode scalar value to a particular code unit sequence for a given encoding form, one can derive the original Unicode scalar value unambiguously from that code unit sequence.
- The mapping of the set of Unicode scalar values to the set of code unit sequences for a Unicode encoding form is not *onto*. In other words for any given encoding form, there exist code unit sequences that have no associated Unicode scalar value.

DRAFT

- To ensure that the mapping for a Unicode encoding form is one-to-one, *all* Unicode scalar values, including those corresponding to noncharacter code points and unassigned code points, must be mapped to unique code unit sequences. Note that this requirement does not extend to high-surrogate and low-surrogate code points, which are excluded by definition from the set of Unicode scalar values.

D29a *Unicode string:* A code unit sequence containing code units of a particular Unicode encoding form.

- In the rawest form, Unicode strings may be implemented simply as arrays of the appropriate integral data type, consisting of a sequence of code units lined up one immediately after the other.
- A single Unicode string must contain only code units from a single Unicode encoding form. It is not permissible to mix forms within a string.

D29b *Unicode 8-bit string:* A Unicode string containing only UTF-8 code units.

D29c *Unicode 16-bit string:* A Unicode string containing only UTF-16 code units.

D29d *Unicode 32-bit string:* A Unicode string containing only UTF-32 code units.

D30 *Ill-formed:* A Unicode code unit sequence that purports to be in a Unicode encoding form is called *ill-formed* if and only if it does not follow the specification of that Unicode encoding form.

- Any code unit sequence that would correspond to a code point outside the defined range of Unicode scalar values would, for example, be ill-formed.
- UTF-8 has some strong constraints on the possible byte ranges for leading and trailing bytes. A violation of those constraints would produce a code unit sequence that could not be mapped to a Unicode scalar value, resulting in an ill-formed code unit sequence.

D30a *Well-formed:* A Unicode code unit sequence that purports to be in a Unicode encoding form is called *well-formed* if and only if it *does* follow the specification of that Unicode encoding form.

- A Unicode code unit sequence that consists entirely of a sequence of well-formed Unicode code unit sequences (all of the same Unicode encoding form) is itself a well-formed Unicode code unit sequence.

D30b *Well-formed UTF-8 code unit sequence:* A well-formed Unicode code unit sequence of UTF-8 code units.

D30c *Well-formed UTF-16 code unit sequence:* A well-formed Unicode code unit sequence of UTF-16 code units.

D30d *Well-formed UTF-32 code unit sequence:* A well-formed Unicode code unit sequence of UTF-32 code units.

D30e *In a Unicode encoding form:* A Unicode string is said to be *in* a particular Unicode encoding form if and only if it consists of a well-formed Unicode code unit sequence of that Unicode encoding form.

- A Unicode string consisting of a well-formed UTF-8 code unit sequence is said to be *in UTF-8*. Such a Unicode string is referred to as a *valid UTF-8 string*, or a *UTF-8 string* for short.
- A Unicode string consisting of a well-formed UTF-16 code unit sequence is said to be *in UTF-16*. Such a Unicode string is referred to as a *valid UTF-16 string*, or a *UTF-16 string* for short.

DRAFT

- A Unicode string consisting of a well-formed UTF-32 code unit sequence is said to be *in UTF-32*. Such a Unicode string is referred to as a *valid UTF-32 string*, or a *UTF-32 string* for short.

Unicode strings need not contain well-formed code unit sequences under all conditions. This is equivalent to saying that a particular Unicode string need not be *in* a Unicode encoding form.

- For example, it is perfectly reasonable to talk about an operation that takes the two Unicode 16-bit strings, <004D D800> and <DF02 004D>, each of which contains an ill-formed UTF-16 code unit sequence, and concatenates them to form another Unicode string <004D D800 DF02 004D>, which contains a well-formed UTF-16 code unit sequence. The first two Unicode strings are not *in* UTF-16, but the resultant Unicode string is.
- As another example, the code unit sequence <C0 80 61 F3> is a Unicode 8-bit string, but does not consist of a well-formed UTF-8 code unit sequence. That code unit sequence could not result from the specification of the UTF-8 encoding form, and is thus ill-formed. (The same code unit sequence could, of course, be well-formed in the context of some other character encoding standard using 8-bit code units, such as ISO/IEC 8859-1, or vendor code pages.)

If, on the other hand, a Unicode string *purports* to be *in* a Unicode encoding form, then it must contain only a well-formed code unit sequence. If there is an ill-formed code unit sequence in a source Unicode string, then a conformant process that verifies that the Unicode string is in a Unicode encoding form must reject the ill-formed code unit sequence. (See conformance clause C12.) For more information, see *Section 2.7, Unicode Strings*.

Table 3-4 gives examples that summarize the three Unicode encoding forms.

Table 3-4. Examples of Unicode Encoding Forms

Code Point	Encoding Form	Code Unit Sequence
U+004D	UTF-32	0000004D
	UTF-16	004D
	UTF-8	4D
U+0430	UTF-32	00000430
	UTF-16	0430
	UTF-8	D0 B0
U+4E8C	UTF-32	00004E8C
	UTF-16	4E8C
	UTF-8	E4 BA 8C
U+10302	UTF-32	00010302
	UTF-16	D800 DF02
	UTF-8	F0 90 8C 82

UTF-32

D31 *UTF-32 encoding form:* The Unicode encoding form which assigns each Unicode scalar value to a single unsigned 32-bit code unit with the same numeric value as the Unicode scalar value.

- In UTF-32, the code point sequence <004D, 0430, 4E8C, 10302> is represented as <0000004D 00000430 00004E8C 00010302>.
- Because surrogate code points are not included in the set of Unicode scalar values, UTF-32 code units in the range $0000D800_{16}..0000DFFF_{16}$ are ill-formed.

DRAFT

- Any UTF-32 code unit greater than $0010FFFF_{16}$ is ill-formed.

For a discussion of the relationship between UTF-32 and UCS-4 encoding form defined in ISO/IEC 10646, see *Section C.2, Encoding Forms in ISO/IEC 10646*.

D32 [Superseded]

D33 [Incorporated in other definitions]

D34 [Incorporated in other definitions]

UTF-16

D35 *UTF-16 encoding form:* The Unicode encoding form which assigns each Unicode scalar value in the ranges U+0000..U+D7FF and U+E000..U+FFFF to a single unsigned 16-bit code unit with the same numeric value as the Unicode scalar value, and which assigns each Unicode scalar value in the range U+10000..U+10FFFF to a surrogate pair, according to *Table 3-5*.

- In UTF-16, the code point sequence <004D, 0430, 4E8C, 10302> is represented as <004D 0430 4E8C D800 DF02>, where <D800 DF02> corresponds to U+10302.
- Because surrogate code points are not Unicode scalar values, isolated UTF-16 code units in the range $D800_{16}$..$DFFF_{16}$ are ill-formed.

Table 3-5 specifies the bit distribution for the UTF-16 encoding form. Note that for Unicode scalar values equal to or greater than U+10000, UTF-16 uses surrogate pairs. Calculation of the surrogate pair values involves subtraction of 10000_{16}, to account for the starting offset to the scalar value. ISO/IEC 10646 specifies an equivalent UTF-16 encoding form. For details, see *Section C.3, UCS Transformation Formats*.

Table 3-5. UTF-16 Bit Distribution

Scalar Value	**UTF-16**
xxxxxxxxxxxxxxxx	xxxxxxxxxxxxxxxx
000uuuuuxxxxxxxxxxxxxxxx	110110wwwwxxxxxx 110111xxxxxxxxxx

Where wwww = uuuuu - 1.

[Check table footnote above. It's not currently attached to the table. JDA 10-25-04]

UTF-8

D36 *UTF-8 encoding form:* The Unicode encoding form which assigns each Unicode scalar value to an unsigned byte sequence of one to four bytes in length, as specified in *Table 3-6*.

- In UTF-8, the code point sequence <004D, 0430, 4E8C, 10302> is represented as <4D D0 B0 E4 BA 8C F0 90 8C 82>, where <4D> corresponds to U+004D, <D0 B0> corresponds to U+0430, <E4 BA 8C> corresponds to U+4E8C, and <F0 90 8C 82> corresponds to U+10302.
- Any UTF-8 byte sequence that does not match the patterns listed in *Table 3-7* is ill-formed.
- Before the Unicode Standard, Version 3.1, the problematic "non-shortest form" byte sequences in UTF-8 were those where BMP characters could be represented in more than one way. These sequences are ill-formed, because they are not allowed by *Table 3-7*.

DRAFT

- Because surrogate code points are not Unicode scalar values, any UTF-8 byte sequence that would otherwise map to code points D800..DFFF is ill-formed.

Table 3-6 specifies the bit distribution for the UTF-8 encoding form, showing the ranges of Unicode scalar values corresponding to one-, two-, three-, and four-byte sequences. For a discussion of the difference in the formulation of UTF-8 in ISO/IEC 10646, see *Section C.3, UCS Transformation Formats*.

Table 3-6. UTF-8 Bit Distribution

Scalar Value	1st Byte	2nd Byte	3rd Byte	4th Byte
00000000 0xxxxxxx	0xxxxxxx			
00000yyy yyxxxxxx	110yyyyy	10xxxxxx		
zzzzyyyy yyxxxxxx	1110zzzz	10yyyyyy	10xxxxxx	
000uuuuu zzzzyyyy yyxxxxxx	11110uuu	10uuzzzz	10yyyyyy	10xxxxxx

Table 3-7 lists all of the byte sequences that are well-formed in UTF-8. A range of byte values such as A0..BF indicates that any byte from A0 to BF (inclusive) is well-formed in that position. Any byte value outside of the ranges listed is ill-formed. For example:

- The byte sequence <C0 AF> is *ill-formed*, because C0 is not well-formed in the "1st Byte" column.
- The byte sequence <E0 9F 80> is *ill-formed*, because in the row where E0 is well-formed as a first byte, 9F is not well-formed as a second byte.
- The byte sequence <F4 80 83 92> is *well-formed*, because every byte in that sequence matches a byte range in a row of the table (the last row).

Table 3-7. Well-Formed UTF-8 Byte Sequences

Code Points	1st Byte	2nd Byte	3rd Byte	4th Byte
U+0000..U+007F	00..7F			
U+0080..U+07FF	C2..DF	80..BF		
U+0800..U+0FFF	E0	***A0***..BF	80..BF	
U+1000..U+CFFF	E1..EC	80..BF	80..BF	
U+D000..U+D7FF	ED	80..***9F***	80..BF	
U+E000..U+FFFF	EE..EF	80..BF	80..BF	
U+10000..U+3FFFF	F0	***90***..BF	80..BF	80..BF
U+40000..U+FFFFF	F1..F3	80..BF	80..BF	80..BF
U+100000..U+10FFFF	F4	80..***8F***	80..BF	80..BF

Cases where a trailing byte range is not 80..BF are in bold italic to draw attention to them. These occur only in the second byte of a sequence.

[Above table footnote has problems to check also re attachment below table. JDA 10-25-04]

As a consequence of the well-formedness conditions specified in *Table 3-7*, the following byte values are disallowed in UTF-8: C0–C1, F5–FF.

Encoding Form Conversion

D37 *Encoding form conversion:* A conversion defined directly between the code unit sequences of one Unicode encoding form and the code unit sequences of another Unicode encoding form.

DRAFT

- In implementations of the Unicode Standard, a typical API will logically convert the input code unit sequence into Unicode scalar values (code points), and then convert those Unicode scalar values into the output code unit sequence. However, proper analysis of the encoding forms makes it possible to convert the code units directly, thereby obtaining the same results but with a more efficient process.
- A conformant encoding form conversion will treat any ill-formed code unit sequence as an error condition. (See conformance clause C12a.) This guarantees that it will neither interpret nor emit an ill-formed code unit sequence. Any implementation of encoding form conversion must take this requirement into account, because an encoding form conversion implicitly involves a verification that the Unicode strings being converted do, in fact, contain well-formed code unit sequences.

3.10 Unicode Encoding Schemes

D38 *Unicode encoding scheme:* A specified byte serialization for a Unicode encoding form, including the specification of the handling of a byte order mark (BOM), if allowed.

- For historical reasons, the Unicode encoding schemes are also referred to as *Unicode* (or *UCS*) *transformation formats* (UTF). That term is, however, ambiguous between its usage for encoding forms and encoding schemes.

The Unicode Standard supports seven encoding schemes. This section presents the formal definition of each of these encoding schemes.

D39 *UTF-8 encoding scheme:* The Unicode encoding scheme that serializes a UTF-8 code unit sequence in exactly the same order as the code unit sequence itself.

- In the UTF-8 encoding scheme, the UTF-8 code unit sequence <4D D0 B0 E4 BA 8C F0 90 8C 82> is serialized as <4D D0 B0 E4 BA 8C F0 90 8C 82>.
- Because the UTF-8 encoding form already deals in ordered byte sequences, the UTF-8 encoding scheme is trivial. The byte ordering is already obvious and completely defined by the UTF-8 code unit sequence itself. The UTF-8 encoding scheme is defined merely for completeness of the Unicode character encoding model.
- While there is obviously no need for a byte order signature when using UTF-8, there are occasions when processes convert UTF-16 or UTF-32 data containing a byte order mark into UTF-8. When represented in UTF-8, the byte order mark turns into the byte sequence <EF BB BF>. Its usage at the beginning of a UTF-8 data stream is neither required nor recommended by the Unicode Standard, but its presence does not affect conformance to the UTF-8 encoding scheme. Identification of the <EF BB BF> byte sequence at the beginning of a data stream can, however, be taken as near-certain indication that the data stream is using the UTF-8 encoding scheme.

D40 *UTF-16BE encoding scheme:* The Unicode encoding scheme that serializes a UTF-16 code unit sequence as a byte sequence in big-endian format.

- In UTF-16BE, the UTF-16 code unit sequence <004D 0430 4E8C D800 DF02> is serialized as <00 4D 04 30 4E 8C D8 00 DF 02>.
- In UTF-16BE, an initial byte sequence <FE FF> is interpreted as U+FEFF ZERO WIDTH NO-BREAK SPACE.

D41 *UTF-16LE encoding scheme:* The Unicode encoding scheme that serializes a UTF-16 code unit sequence as a byte sequence in little-endian format.

DRAFT

- In UTF-16LE, the UTF-16 code unit sequence <004D 0430 4E8C D800 DF02> is serialized as <4D 00 30 04 8C 4E 00 D8 02 DF>.
- In UTF-16LE, an initial byte sequence <FF FE> is interpreted as U+FEFF ZERO WIDTH NO-BREAK SPACE.

D42 *UTF-16 encoding scheme:* The Unicode encoding scheme that serializes a UTF-16 code unit sequence as a byte sequence in either big-endian or little-endian format.

- In the UTF-16 encoding scheme, the UTF-16 code unit sequence <004D 0430 4E8C D800 DF02> is serialized as <FE FF 00 4D 04 30 4E 8C D8 00 DF 02> or <FF FE 4D 00 30 04 8C 4E 00 D8 02 DF> or <00 4D 04 30 4E 8C D8 00 DF 02>.
- In the UTF-16 encoding scheme, an initial byte sequence corresponding to U+FEFF is interpreted as a *byte order mark* (BOM); it is used to distinguish between the two byte orders. An initial byte sequence <FE FF> indicates big-endian order, and an initial byte sequence <FF FE> indicates little-endian order. The BOM is not considered part of the content of the text.
- The UTF-16 encoding scheme may or may not begin with a BOM. However, when there is no BOM, and in the absence of a higher-level protocol, the byte order of the UTF-16 encoding scheme is big-endian.

Table 3-8 gives examples that summarize the three Unicode encoding schemes for the UTF-16 encoding form.

Table 3-8. Summary of UTF-16BE, UTF-16LE, and UTF-16

Code Unit Sequence	Encoding Scheme	Byte Sequence(s)
004D	UTF-16BE	00 4D
	UTF-16LE	4D 00
	UTF-16	FE FF 00 4D FF FE 4D 00 00 4D
0430	UTF-16BE	04 30
	UTF-16LE	30 04
	UTF-16	FE FF 04 30 FF FE 30 04 04 30
4E8C	UTF-16BE	4E 8C
	UTF-16LE	8C 4E
	UTF-16	FE FF 4E 8C FF FE 8C 4E 4E 8C
D800 DF02	UTF-16BE	D8 00 DF 02
	UTF-16LE	00 D8 02 DF
	UTF-16	FE FF D8 00 DF 02 FF FE 00 D8 02 DF D8 00 DF 02

D43 *UTF-32BE encoding scheme:* The Unicode encoding scheme that serializes a UTF-32 code unit sequence as a byte sequence in big-endian format.

- In UTF-32BE, the UTF-32 code unit sequence <0000004D 00000430 00004E8C 00010302> is serialized as <00 00 00 4D 00 00 04 30 00 00 4E 8C 00 01 03 02>.
- In UTF-32BE, an initial byte sequence <00 00 FE FF> is interpreted as U+FEFF ZERO WIDTH NO-BREAK SPACE.

DRAFT

D44 *UTF-32LE encoding scheme:* The Unicode encoding scheme that serializes a UTF-32 code unit sequence as a byte sequence in little-endian format.

- In UTF-32LE, the UTF-32 code unit sequence <0000004D 00000430 00004E8C 00010302> is serialized as <4D 00 00 00 30 04 00 00 8C 4E 00 00 02 03 01 00>.
- In UTF-32LE, an initial byte sequence <FF FE 00 00> is interpreted as U+FEFF ZERO WIDTH NO-BREAK SPACE.

D45 *UTF-32 encoding scheme:* The Unicode encoding scheme that serializes a UTF-32 code unit sequence as a byte sequence in either big-endian or little-endian format.

- In the UTF-32 encoding scheme, the UTF-32 code unit sequence <0000004D 00000430 00004E8C 00010302> is serialized as <00 00 FE FF 00 00 00 4D 00 00 04 30 00 00 4E 8C 00 01 03 02> or <FF FE 00 00 4D 00 00 00 30 04 00 00 8C 4E 00 00 02 03 01 00> or <00 00 00 4D 00 00 04 30 00 00 4E 8C 00 01 03 02>.
- In the UTF-32 encoding scheme, an initial byte sequence corresponding to U+FEFF is interpreted as a *byte order mark* (BOM); it is used to distinguish between the two byte orders. An initial byte sequence <00 00 FE FF> indicates big-endian order, and an initial byte sequence <FF FE 00 00> indicates little-endian order. The BOM is not considered part of the content of the text.
- The UTF-32 encoding scheme may or may not begin with a BOM. However, when there is no BOM, and in the absence of a higher-level protocol, the byte order of the UTF-32 encoding scheme is big-endian.

Table 3-9 gives examples that summarize the three Unicode encoding schemes for the UTF-32 encoding form.

Table 3-9. Summary of UTF-32BE, UTF-32LE, and UTF-32

Code Unit Sequence	Encoding Scheme	Byte Sequence(s)
0000004D	UTF-32BE	00 00 00 4D
	UTF-32LE	4D 00 00 00
	UTF-32	00 00 FE FF 00 00 00 4D FF FE 00 00 4D 00 00 00 00 00 00 4D
00000430	UTF-32BE	00 00 04 30
	UTF-32LE	30 04 00 00
	UTF-32	00 00 FE FF 00 00 04 30 FF FE 00 00 30 04 00 00 00 00 04 30
00004E8C	UTF-32BE	00 00 4E 8C
	UTF-32LE	8C 4E 00 00
	UTF-32	00 00 FE FF 00 00 4E 8C FF FE 00 00 8C 4E 00 00 00 00 4E 8C
00010302	UTF-32BE	00 01 03 02
	UTF-32LE	02 03 01 00
	UTF-32	00 00 FE FF 00 01 03 02 FF FE 00 00 02 03 01 00 00 01 03 02

Note that the terms *UTF-8*, *UTF-16*, and *UTF-32*, when used unqualified, are ambiguous between their sense as Unicode encoding forms or Unicode encoding schemes. For UTF-8, this ambiguity is usually innocuous, because the UTF-8 encoding scheme is trivially derived from the byte sequences defined for the UTF-8 encoding form. However, for UTF-16 and UTF-32, the ambiguity is more problematical. As encoding forms, UTF-16 and

DRAFT

UTF-32 refer to code units in memory; there is no associated byte orientation, and a BOM is never used. As encoding schemes, UTF-16 and UTF-32 refer to serialized bytes, as for streaming data or in files; they may have either byte orientation, and a BOM may be present.

When the usage of the short terms "UTF-16" or "UTF-32" might be misinterpreted, and where a distinction between their use as referring to Unicode encoding forms or to Unicode encoding schemes is important, the full terms, as defined in this chapter of the Unicode Standard, should be used. For example, use *UTF-16 encoding form* or *UTF-16 encoding scheme*. They may also be abbreviated to *UTF-16 CEF* or *UTF-16 CES*, respectively.

When converting between different encoding schemes, extreme care must be taken in handling any initial byte order marks. For example, if one converted a UTF-16 byte serialization with an initial byte order mark to a UTF-8 byte serialization, converting the byte order mark to <EF BB BF> in the UTF-8 form, the <EF BB BF> would now be ambiguous as to its status as a byte order mark (from its source) or as an initial *zero width no-break space*. If the UTF-8 byte serialization were then converted to UTF-16BE and the initial <EF BB BF> were converted to <FE FF>, the interpretation of the U+FEFF character would have been modified by the conversion. This would be nonconformant according to conformance clause C10, because the change between byte serializations would have resulted in modification of the interpretation of the text. This is one reason why the use of initial <EF BB BF> as a signature on UTF-8 byte sequences is not recommended by the Unicode Standard.

3.11 Canonical Ordering Behavior

This section provides a formal statement of canonical ordering behavior, which determines, for the purposes of interpretation, which combining character sequences are to be considered equivalent. A precise definition of equivalence is required, so that text containing combining character sequences can be created and interchanged in a predictable way.

When combining sequences contain multiple combining characters, different sequences can contain the same characters, but in a different order. Under certain circumstances two such sequences may be equivalent, even though they differ in the order of the combining characters.

Canonical ordering is a process of specifying a defined order for sequences of combining marks, whereby it is possible to determine definitively which sequences are equivalent and which are not.

Canonical ordering behavior, and more specifically, *canonical ordering*, is a required part of the normative specification of *normalization* for the Unicode Standard. See Unicode Standard Annex #15, "Unicode Normalization Forms."

Canonical ordering is also a required part of the separate standard, Unicode Technical Standard #10, "Unicode Collation Algorithm."

Application of Combining Marks

There are a number of principles in the Unicode Standard regarding the application of combining marks. These principles are listed in this section, with an indication of which are considered to be normative and which are considered to be guidelines.

In particular, guidelines for rendering of combining marks in conjunction with other characers should be considered as appropriate for defining default rendering behavior, in the absence of more specific information about rendering. It is often the case that combin-

DRAFT

ing marks in complex scripts, or even particular, general-use nonspacing marks will have rendering requirements that depart significantly from the general guidelines. Rendering processes should, as appropriate, make use of available information about specific typographic practices and conventions, in order to produce best rendering of text.

To help in the clarification of the principles regarding the application of combining marks, a distinction is made between *dependence* and *graphical application*.

D45a *Dependence:* A combining mark is said to *depend* on its associated base character.

- The associated base character is the base character in the combining character sequence that a combining mark is part of.
- A combining mark in a defective combining character sequence has no associated base character, and thus cannot be said to depend on any particular base character. This is one of the reasons why fallback processing is required for defective combining character sequences.
- Dependence concerns *all* combining marks, including spacing marks and combining marks that have no visible display.

D45b *Graphical application:* A nonspacing mark is said to *apply* to its associated grapheme base.

- The associated grapheme base is the grapheme base in the grapheme cluster that a nonspacing mark is part of.
- A nonspacing mark in a defective combining character sequence is not part of a grapheme cluster, and is subject to the same kinds of fallback processing as for any defective combining character sequence.
- Graphic application concerns visual rendering issues, and thus is an issue for nonspacing marks that have visible glyphs. Those glyphs interact, in rendering, with their grapheme base.

Throughout the text of the standard, whenever the situation is clear, discussion of combining marks often simply talks about combining marks "applying" to their base. In the prototypical case, often illustrated, of a nonspacing accent mark applying to a single base character letter, this simplification is not problematical, because the nonspacing mark both depends (notionally) on its base character and simultaneously applies (graphically) to its grapheme base, affecting its display. The finer distinctions are needed when dealing with the edge cases, such as combining marks that have no display glyph, graphical application of nonspacing marks to Korean syllables, and the behavior of spacing combining marks.

The distinction made here between notional dependence and graphical application does not preclude spacing marks or even sequences of base characters from having effects on neighboring characters in rendering. Thus, spacing forms of dependent vowels (*matras*) in Indic scripts, may trigger particular kinds of conjunct formation, or may be repositioned in ways that influence the rendering of other characters. (See *Chapter 9, South Asian Scripts-I*, for many examples.) Similarly, sequences of base characters may also form ligatures in rendering. (See "Cursive Connection and Ligatures" in *Section 16.2, Layout Controls*.)

The following listing specifies the principles regarding application of combining marks.

P1 ***[Normative] Combining character order: Combining characters follow the base character on which they depend.***

- This principle follows from the definition of a combining character sequence.

DRAFT

- Thus the character sequence <U+0061 "a" LATIN SMALL LETTER A, U+0308 "◌̈" COMBINING DIAERESIS, U+0075 "u" LATIN SMALL LETTER U> is unambiguously interpreted (and displayed) as "äu", not "aü"

P2 ***[Guideline] Inside-out application. Nonspacing marks with the same combining class are generally positioned graphically outward from the grapheme base to which they apply.***

- The most numerous and important instances of this principle involve nonspacing marks applied either directly above or below a grapheme base.
- In a sequence of two nonspacing marks above a grapheme base, the first nonspacing mark is placed directly above the grapheme base, and the second is then placed above the first nonspacing mark.
- In a sequence of two nonspacing marks below a grapheme base, the first nonspacing mark is placed directly below the grapheme base, and the second is then placed below the first nonspacing mark.
- This rendering behavior for nonspacing marks can be generalized to sequences of any length, although practical considerations usually limit such sequences to no more than two or three marks above and/or below a grapheme base.
- The principle of inside-out application is also referred to as *default stacking behavior* for nonspacing marks.

P3 ***[Guideline] Side-by-side application. Notwithstanding the principle of inside-out application, some specific nonspacing marks may override the default stacking behavior and are positioned side-by-side over (or under) a grapheme base, rather than stacking vertically.***

- Such side-by-side positioning may reflect language-specific orthographic rules, such as for Vietnamese diacritics and tone marks, or for polytonic Greek breathing and accent marks. For examples, see *Section 2.11, Combining Characters*.
- When positioned side-by-side, the visual rendering order of a sequence of non-spacing marks reflects the dominant order of the script with which they are used. Thus in Greek, the first non-spacing mark in such a sequence will be positioned to the left side above a grapheme base, and the second to the right side above the grapheme base. In Hebrew, the opposite positioning is used for side-by-side placement.

P4 ***[Guideline] Traditional typographical behavior will sometimes override the default placement or rendering of nonspacing marks.***

- Because of typographical conflict with the descender of a base character, a combining comma below placed on a lowercase "g" is traditionally rendered as if it were an inverted comma above. See *Section 7.1, Latin*.
- Because of typographical conflict with the ascender of a base chracter, a combining háček (caron) is traditionally rendered as an apostrophe when placed, for example, on a lowercase "d". See *Section 7.1, Latin*.
- The relative placement of vowel marks in Arabic cannot be predicted by default stacking behavior alone, but depends on traditional rules of Arabic typography. See *Section 8.2, Arabic*.

[Confirm 8.2 Arabic cross ref above once we have the example in for the ed note that we need an example of non-default placement of vowel marks. JDA per Asmus 10-1-05]

DRAFT

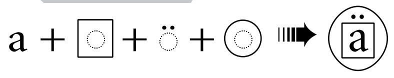
P5 ***[Normative] Nondistinct order. Nonspacing marks with different, non-zero combining classes may occur in different orders without affecting either the visual display of a combining character sequence or the interpretation of that sequence.***

- For example, if one nonspacing mark occurs above a grapheme base and another nonspacing mark occurs below, they will have distinct combining classes, and the order in which they occur in the combining character sequence does not matter for the display or interpretation of the resulting grapheme cluster.
- Inserting a *combining grapheme joiner* between two combining marks with nondistinct order prevents their canonical reordering. For more information, see "Combining Grapheme Joiner" in *Section 16.2, Layout Controls*.
- The introduction of the combining class for characters and its use in canonical ordering in the standard is to precisely define canonical equivalence, and thereby to clarify exactly which such alternate sequences must be considered as identical for display and interpretation.
- In cases of nondistinct order, the order of combining marks has no linguistic significance. The order does not reflect how "closely bound" they are to the base. After canonical reordering, the order may no longer reflect the typed-in sequence. Rendering systems should be prepared to deal with common type-in sequences and with canonically reordered sequences.

P6 ***[Guideline] Enclosing marks surround their grapheme base and any intervening nonspacing marks.***

- This implies that enclosing marks successively surround previous enclosing marks. See *Figure 3-1*.

Figure 3-1. Enclosing Marks



- Dynamic application of enclosing marks, particularly sequences of enclosing marks, is beyond the capability of most fonts and simple rendering processes. so it is not unexpected to find fallback rendering in cases such as that illustrated in *Figure 3-1*.

P7 ***[Guideline] Double diacritic nonspacing marks, such as U+0360 COMBINING DOUBLE TILDE, apply to their grapheme base, but are intended to be rendered with glyphs that encompass a following grapheme base as well. See Figure 7-7 for an example.***

- Because such double diacritic display spans combinations of elements which would otherwise be considered grapheme clusters, the support of double diacritics in rendering may involve special handling for cursor placement and text selection.

P8 ***[Guideline] When double diacritic nonspacing marks interact with normal nonspacing marks in a grapheme cluster, they "float" to the outermost layer of the stack of rendered marks (either above or below). See Figure 7-8 for an example.***

- This behavior can be conceived of as a kind of looser binding of such double diacritics to their bases. In effect, all other nonspacing marks are applied first, and then the double diacritic will span the resulting stacks.

DRAFT

- Double diacritic nonspacing marks are also given a very high combining class, so that in canonical order they appear at or near the end of any combining character sequence.
- The interaction of enclosing marks and double diacritics is not well-defined graphically. It is unlikely that most fonts or rendering processes could handle combinations of these felicitously. It is not recommended to use combinations of these together in the same grapheme cluster.

Combining Marks and Korean Syllables. When a grapheme cluster comprises a Korean syllable, a combining mark applies to that entire syllable. For example, in the following sequence the *grave* is applied to the entire Korean syllable, not just to the last jamo:

> U+1100 ᄀ *choseong kiyeok* + U+1161 ᅡ *jungseong a* + U+0300 ◌̀ *grave* →
> 가̀

If the combining mark in question is an *enclosing* combining mark, then it would enclose the entire Korean syllable, rather than the last jamo in it:

> U+1100 ᄀ *choseong kiyeok* + U+1161 ᅡ *jungseong a* + U+20DD ◌⃝
> *enclosing circle* → 가⃝

This treatment of the application of combining marks with respect to Korean syllables follows from the implications of canonical equivalence. It should be noted, however, that older implementations may have supported the application of an enclosing combining mark to an entire Indic consonant conjunct or to a sequence of grapheme clusters linked together by combining grapheme joiners. Such an approach has a number of technical problems and leads to interoperability defects, so it is strongly recommended that implementations do not follow it.

For more information on the recommended use of the combining grapheme joiner, see the subsection "Combining Grapheme Joiner" in *Section 16.2, Layout Controls*. For more discussion regarding the application of combining marks in general, see *Section 7.9, Combining Marks*.

Combining Classes

Each character in the Unicode Standard has a combining class associated with it. The combining class is a numerical value used by the canonical ordering algorithm to determine which sequences of combining marks are to be considered canonically equivalent and which are not. Canonical equivalence is the criterion for whether two alternate sequences are considered identical for interpretation.

D46 *Combining class:* A numeric value in the range 0..255 given to each Unicode code point, formally defined as the property Canonical_Combining_Class.

- The combining class for each encoded character in the standard is specified in the file UnicodeData.txt in the Unicode Character Database. Any code point not listed in that data file defaults to [Canonical_Combining_Class = 0] (or [ccc = 0] for short).
- An extracted listing of combining classes, sorted by numeric value, is provided in the file DerivedCombiningClass.txt in the Unicode Character Database.
- Only combining marks have a combining class other than zero. Almost all combining marks with a class other than zero are also nonspacing marks, but there are a few exceptions. And not all nonspacing marks have a non-zero combining class. So while the correlation between [ccc ≠ 0] and [gc = Mn] is close, it is not exact, and implementations should not depend on the two concepts being identical.

DRAFT

D46a *Fixed position class:* A subset of the range of numeric values for combining classes, specifically any value in the range 10..199.

- Fixed position classes are assigned to a small number of Hebrew, Arabic, Syriac, Telugu, Thai, Lao, and Tibetan combining marks whose position was conceived of as occurring in a fixed position with respect to their grapheme base, regardless of any other combining mark which might also apply to that grapheme base.
- Not all Arabic vowel points or Indic matras are given fixed position classes. The existence of fixed position classes in the standard is an historical artifact of an earlier stage in its development, prior to the formal standardization of the Unicode Normalization Forms.

D46b *Typographic interaction:* Graphical application of one nonspacing mark in a position relative to a grapheme base that is already occupied by another nonspacing mark, so that some rendering adjustment must be done (such as default stacking or side-by-side placement) to avoid illegible overprinting or crashing of glyphs.

The assignment of combining class values for Unicode characters was originally done with the goal in mind of defining distinct numeric values for each group of nonspacing marks that would typographically interact. Thus all generic nonspacing marks above are given the value [ccc = 230], while all generic nonspacing marks below are given the value [ccc = 220]. Smaller numbers of nonspacing marks which tend to sit on one "shoulder" or another of a grapheme base, or which may actually be attached to the grapheme base itself when applied, have their own combining classes.

When assigned this way, canonical ordering assures that, in general, alternate sequences of combining characters that typographically interact will not be canonically equivalent, whereas alternate sequences of combining characters that do *not* typographically interact *will* be canonically equivalent.

This is roughly correct for the normal cases of detached, generic nonspacing marks placed above and below base letters. However, the ramifications of complex rendering for many scripts ensure that there are always some edge cases where there may be typographic interaction between combining marks of distinct combining classes. This has turned out to be particularly true for some of the fixed position classes for Hebrew and Arabic, for which a distinct combining class is no guarantee that there will be no typographic interaction for rendering.

Because of these considerations, particular combining class values should only be taken as a guideline regarding issues of typographic interaction of combining marks.

The only *normative* use of combining class values is as input to the canonical ordering algorithm, where they are used to normatively distinguish between sequences of combining marks that are canonically equivalent and those which are not.

Canonical Ordering

The canonical ordering of a decomposed character sequence results from a sorting process that acts on each sequence of combining characters according to their combining class. The canonical order of character sequences does *not* imply any kind of linguistic correctness or linguistic preference for ordering of combining marks in sequences. See the information on rendering combining marks in *Section 5.13, Rendering Nonspacing Marks*, for more information. Characters with combining class zero never reorder relative to other characters, so the amount of work in the algorithm depends on the number of non-class-zero characters in a row. An implementation of this algorithm will be extremely fast for typical text.

DRAFT

The algorithm described here represents a logical description of the process. Optimized algorithms can be used in implementations as long as they are equivalent—that is, as long as they produce the same result. This algorithm is not tailorable; higher-level protocols shall not specify different results.

More explicitly, the canonical ordering of a decomposed character sequence D results from the following algorithm.

R9 ***For each character x in D, let p(x) be the combining class of x.***

R10 ***Whenever any pair (A, B) of adjacent characters in D is such that p(B) ‡ 0 & p(A) > p(B), exchange those characters.***

R11 ***Repeat step R2 until no exchanges can be made among any of the characters in D.***

Sample combining classes for this discussion are listed in *Table 3-10*.

Table 3-10. Sample Combining Classes

Combining Class	Abbreviation	Code	Unicode Name
0	a	U+0061	latin small letter a
220	underdot	U+0323	combining dot below
230	diaeresis	U+0308	combining diaeresis
230	breve	U+0306	combining breve
0	a-underdot	U+1EA1	latin small letter a with dot below
0	a-diaeresis	U+00E4	latin small letter a with diaeresis
0	a-breve	U+0103	latin small letter a with breve

Because *underdot* has a lower combining class than *diaeresis*, the algorithm will return the *a*, then the *underdot*, then the *diaeresis*. The sequence *a + underdot + diaeresis* is already in the final order, and so is not rearranged by the algorithm. The sequence in the opposite order, *a + diaeresis + underdot*, is rearranged by the algorithm.

a + underdot + diaeresis	í	*a + underdot + diaeresis*
a + diaeresis + underdot	í	*a + underdot + diaeresis*

However, because *diaeresis* and *breve* have the same combining class (because they interact typographically), they do not rearrange.

a + breve + diaeresis	õ	*a + diaeresis + breve*
a + diaeresis + breve	õ	*a + breve + diaeresis*

Applying the algorithm gives the results shown in *Table 3-11*.

Table 3-11. Canonical Ordering Results

Original	Decompose	Sort	Result
a-diaeresis + underdot	a + diaeresis + underdot	a + underdot + diaeresis	a + underdot + diaeresis
a + diaeresis + underdot		a + underdot + diaeresis	a + underdot + diaeresis
a + underdot + diaeresis			a + underdot + diaeresis
a-underdot + diaeresis	a + underdot + diaeresis		a + underdot + diaeresis
a-diaeresis + breve	a + diaeresis + breve		a + diaeresis + breve
a + diaeresis + breve			a + diaeresis + breve
a + breve + diaeresis			a + breve + diaeresis
a-breve + diaeresis	a + breve + diaeresis		a + breve + diaeresis

DRAFT

3.12 Conjoining Jamo Behavior

The Unicode Standard contains both a large set of precomposed modern Hangul syllables and a set of conjoining Hangul jamo, which can be used to encode archaic Korean syllable blocks as well as modern Korean syllable blocks. This section describes how to

- Determine the syllable boundaries in a sequence of conjoining jamo characters
- Compose jamo characters into precomposed Hangul syllables
- Determine the canonical decomposition of precomposed Hangul syllables
- Algorithmically determine the names of precomposed Hangul syllables

For more information, see the “Hangul Syllables” and “Hangul Jamo” subsections in *Section 12.5, Hangul*. Hangul syllables are a special case of grapheme clusters.

K25 *Leading consonant:* In the Korean context, a jamo character in the range U+1100..U+1159. Abbreviated as *L*.

- When not occurring in clusters, the term *leading consonant* is equivalent to *syllable-initial character*.

K26 *Choseong:* A sequence of one or more leading consonants.

- In Modern Korean, a *choseong* consists of a single jamo, but in Old Korean, a sequence of more than one leading consonant may occur.
- Equivalent to *syllable-initial cluster*.

K27 *Choseong filler:* U+115F HANGUL CHOSEONG FILLER. Abbreviated as L_f.

- A *choseong filler* stands in for a missing choseong to make a well-formed Korean syllable.

K28 *Vowel:* In the Korean context, a jamo character in the range U+1161..U+11A2. Abbreviated as *V*.

- When not occurring in clusters, the term *vowel* is equivalent to *syllable-peak character*.

K29 *Jungseong:* A sequence of one or more vowels.

- In Modern Korean, a *jungseong* consists of a single jamo, but in Old Korean, a sequence of more than one vowel may occur.
- Equivalent to *syllable-peak cluster*.

K30 *Jungseong filler:* U+1160 HANGUL JUNGSEONG FILLER. Abbreviated as V_f.

- A *jungseong filler* stands in for a missing jungseong to make a well-formed Korean syllable.

K31 *Trailing consonant:* In the Korean context, a jamo character in the range U+11A8..U+11F9. Abbreviated as *T*.

- When not occurring in clusters, the term *trailing consonant* is equivalent to *syllable-final character*.

K32 *Jongseong:* A sequence of one or more trailing consonants.

- In Modern Korean, a *jongseong* consists of a single jamo, but in Old Korean, a sequence of more than one trailing consonant may occur.
- Equivalent to *syllable-final cluster*.

DRAFT

K33 *Precomposed Hangul syllable:* Any of the characters in the range U+AC00..U+D7A3.

K34 *Syllable block:* A sequence of Korean characters that should be grouped into a single square cell for display.

- This is different from a precomposed Hangul syllable, and is meant to include sequences needed for the representation of Old Korean syllables.
- A syllable block may contain a precomposed Hangul syllable *plus* other characters.

Hangul Syllable Boundary Determination

In rendering, a sequence of jamos is displayed as a series of syllable blocks. The following rules specify how to divide up an arbitrary sequence of jamos (including nonstandard sequences) into these syllable blocks.

The precomposed Hangul syllables are of two types: *LV* or *LVT*. In determining the syllable boundaries, the *LV* behave as if they were a sequence of jamo *L V*, and the *LVT* behave as if they were a sequence of jamo *L V T*.

Within any sequence of characters, a syllable break never occurs between the pairs of characters shown in *Table 3-12*. In *Table 3-12* non-opportunities for syllable breaks are shown by "×". Combining marks are shown by the symbol *M*.

In all cases other than those shown in *Table 3-12*, there is a syllable break before and after any jamo or precomposed Hangul syllable. Note that as for other characters, any combining mark between two conjoining jamos prevents the jamos from forming a syllable block.

Table 3-12. Hangul Syllable No-Break Rules

Do Not Break Between		**Examples**
L	L, V, or precomposed Hangul syllable	L × L L × V L × LV L × LVT
V or LV	V or T	V × V V × T LV × V LV × T
T or LVT	T	T × T LVT × T
Jamo or precomposed Hangul syllable	Combining marks	L × M V × M T × M LV × M LVT × M

Even in normalization form NFC, a syllable block may contain a precomposed Hangul syllable in the middle. An example is *L LVT T*. Each well-formed modern Hangul syllable, however, can be represented in the form *L V T?* (that is one *L*, one *V* and optionally one *T*), and consists of a single encoded character in NFC.

For information on the behavior of Hangul compatibility jamo in syllables, see *Section 12.5, Hangul*.

Standard Korean Syllables

K35 *Standard Korean syllable block:* A sequence of one or more *L* followed by a sequence of one or more *V* and a sequence of zero or more *T*.

DRAFT

- A choseong filler may substitute for a missing leading consonant, and a jungseong filler may substitute for a missing vowel.
- Using regular expression notation, a standard Korean syllable block is thus of the form: $(L+ \mid L_f)\ (V+ \mid V_f)\ T*$
- Alternatively, a standard Korean syllable block may be expressed as a sequence of a *choseong* and a *jungseong*, optionally followed by a *jongseong*.

Because of equivalence relations, a standard Korean syllable block may also consist of a precomposed Hangul syllable or a mixture of jamos and a precomposed Hangul syllable character canonically equivalent to a sequence of jamos.

A sequence of jamos which do not all match the regular expression for a standard Korean syllable block can be transformed into a sequence of standard Korean syllable blocks by the correct insertion of *choseong fillers* and *jungseong fillers*. This transformation of a string of text into standard Korean syllables is performed by determining the syllable breaks as explained in the earlier subsection "Hangul Syllable Boundaries," then inserting one or two fillers as necessary to transform each syllable into a standard Korean syllable. Thus:

$$L\ [\text{^}V]\ \text{í}\ L\ V_f\ [\text{^}V]$$

$$[\text{^}L]\ V\ \text{í}\ [\text{^}L]\ L_f\ V$$

$$[\text{^}V]\ T\ \text{í}\ [\text{^}V]\ L_f\ V_f\ T$$

where [^X] indicates a character that is not X, or the absence of a character.

Examples. In *Table 3-13*, the first row shows syllable breaks in a standard sequence, the second row shows syllable breaks in a nonstandard sequence, and the third row shows how the sequence in the second row could be transformed into standard form by inserting fillers into each syllable. Syllable breaks are shown by *middle dots* "·".

Table 3-13. Korean Syllable Break Examples

No.	Sequence		Sequence with Syllable Breaks Marked
1	$LVTLVLVLV_fL_fVL_fV_fT$	í	$LVT \cdot LV \cdot LV \cdot LV_f \cdot L_fV \cdot L_fV_fT$
2	LLTTVVTTVVLLVV	í	LL · TT · VVTT · VV · LLVV
3	LLTTVVTTVVLLVV	í	$LLV_f \cdot L_fV_fTT \cdot L_fVVTT \cdot L_fVV \cdot LLVV$

Hangul Syllable Composition

The following algorithm describes how to take a sequence of canonically decomposed characters D and compose Hangul syllables. Hangul composition and decomposition are summarized here, but for a more complete description, implementers must consult Unicode Standard Annex #15, "Unicode Normalization Forms." Note that, like other non-jamo characters, any combining mark between two conjoining jamos prevents the jamos from composing.

First, define the following constants:

```
SBase  = AC00₁₆
LBase  = 1100₁₆
VBase  = 1161₁₆
TBase  = 11A7₁₆
SCount = 11172
LCount = 19
VCount = 21
TCount = 28
NCount = VCount * TCount
```

DRAFT

1. Iterate through the sequence of characters in D, performing steps 2 through 5.
2. Let *i* represent the current position in the sequence D. Compute the following indices, which represent the ordinal number (zero-based) for each of the components of a syllable, and the index *j*, which represents the index of the last character in the syllable.
   ```
   LIndex = D[i] - LBase
   VIndex = D[i+1] - VBase
   TIndex = D[i+2] - TBase
   j      = i + 2
   ```
3. If either of the first two characters is out of bounds (*LIndex* < 0 OR *LIndex* ≥ *LCount* OR *VIndex* < 0 OR *VIndex* ≥ *VCount*), then increment *i*, return to step 2, and continue from there.
4. If the third character is out of bounds (*TIndex* ≤ 0 or *TIndex* ≥ *TCount*), then it is not part of the syllable. Reset the following:
   ```
   TIndex = 0
   j      = i + 1
   ```
5. Replace the characters D[*i*] through D[*j*] by the Hangul syllable *S*, and set *i* to be *j* + 1.
   ```
   S      = (LIndex * VCount + VIndex) * TCount + TIndex + SBase
   ```

Example. The first three characters are

U+1111	ë	HANGUL CHOSEONG PHIEUPH
U+1171	Ò	HANGUL JUNGSEONG WI
U+11B6	∂	HANGUL JONGSEONG RIEUL-HIEUH

Compute the following indices,

```
LIndex = 17
VIndex = 16
TIndex = 15
```

and replace the three characters by

```
S      = [(17 * 21) + 16] * 28 + 15 + SBase
```
$= \mathtt{D4DB}_{16}$

= 퓛

Hangul Syllable Decomposition

The following algorithm describes the reverse mapping—how to take Hangul syllable *S* and derive the canonical decomposition D. This normative mapping for these characters is equivalent to the canonical mapping in the character charts for other characters.

1. Compute the index of the syllable:
   ```
   SIndex = S - SBase
   ```
2. If *SIndex* is in the range (0 ≤ *SIndex* < *SCount*), then compute the components as follows:
   ```
   L      = LBase + SIndex / NCount
   V      = VBase + (SIndex % NCount) / TCount
   T      = TBase + SIndex % TCount
   ```
   The operators "/" and "%" are as defined in *Section 0.4, Notational Conventions*.
3. If *T* = *TBase*, then there is no trailing character, so replace *S* by the sequence *L V*. Otherwise, there is a trailing character, so replace *S* by the sequence *L V T*.

DRAFT

Example

```
L      = LBase + 17
V      = VBase + 16
T      = TBase + 15
```
$\mathtt{D4DB}_{16} \rightarrow \mathtt{1111}_{16}, \mathtt{1171}_{16}, \mathtt{11B6}_{16}$

Hangul Syllable Name Generation

The character names for Hangul syllables are derived from the decomposition by starting with the string HANGUL SYLLABLE, and appending the short name of each decomposition component in order. (See *Chapter 17, Code Charts*, and Jamo.txt in the Unicode Character Database.) For example, for U+D4DB, derive the decomposition, as shown in the preceding example. It produces the following three-character sequence:

U+1111	HANGUL CHOSEONG PHIEUPH (P)
U+1171	HANGUL JUNGSEONG WI (WI)
U+11B6	HANGUL JONGSEONG RIEUL-HIEUH (LH)

The character name for U+D4DB is then generated as HANGUL SYLLABLE PWILH, using the short name as shown in parentheses above. This character name is a normative property of the character.

3.13 Default Case Algorithms

[This entire section either will be extracted to a new update for UAX #21, or needs to be further rewritten for clarity of definitions and clarification of status of normative and informative material in relation to UAX #29. Ken 10-04]

This section specifies the default operations for case conversion, case detection, and caseless matching. For information about the data sources for case mapping, see *Section 4.2, Case—Normative*. For a general discussion of case mapping operations, see *Section 5.18, Case Mappings*.

The default casing operations are to be used in the absence of tailoring for particular languages and environments. Where a particular environment (such as a Turkish locale) requires tailoring, that can be done without violating conformance.

All the specifications are *logical* specifications; particular implementations can optimize the processes as long as they provide the same results.

Definitions

The full case mappings for Unicode characters are obtained by using the mappings from SpecialCasing.txt *plus* the mappings from UnicodeData.txt, excluding any latter mappings that would conflict. Any character that does not have a mapping in these files is considered to map to itself. In this discussion, the full case mappings of a character C are referred to as default_lower(C), default_title(C), and default_upper(C). The full case folding of a character C is referred to as default_fold(C).

Detection of case and case mapping requires more than just the General Category values (Lu, Lt, Ll). The following definitions are used:

D47 A character C is defined to be *cased* if and only if at least one of following is true for C: uppercase=true, or lowercase=true, or general_category=titlecase_letter.

- The uppercase and lowercase property values are specified in the data file DerivedCoreProperties.txt in the Unicode Character Database.

DRAFT

D47a A character C is defined to be *case-ignorable* if C has the Unicode Property Word_Break=MidLetter as defined in Unicode Standard Annex #29, “Text Boundaries;” or the General Category of C is Nonspacing Mark (Mn), Enclosing Mark (Me), Format Control (Cf), Letter Modifier (Lm), or Symbol Modifier (Sk).

D47b A *case-ignorable sequence* is a sequence of zero or more case-ignorable characters.

D48 A character C is in a particular *casing context* for context-dependent matching if and only if it matches the corresponding specification in *Table 3-14.*

Table 3-14. Context Specification for Casing

Context	**Description**	**Regular Expressions**	
Final_Sigma	C is preceded by a sequence consisting of a cased letter and a case-ignorable sequence, and C is not followed by a sequence consisting of a case ignorable sequence and then a cased letter.	Before C:	\p{cased} (\p{case-ignorable})*
		After C	! ((\p{case-ignorable})* \p{cased})
After_Soft_Dotted	There is a Soft_Dotted character before C, with no intervening character of combining class 0 or 230 (ABOVE).	Before C:	[\p{Soft_Dotted}] ([^\p{cc=230} \p{cc=0}])*
More_Above	C is followed by a character of combining class 230 (ABOVE) with no intervening character of combining class 0 or 230 (ABOVE).	After C:	[^\p{cc=0}]* [\p{cc=230}]
Before_Dot	C is followed by COMBINING DOT ABOVE (U+0307). Any sequence of characters with a combining class that is neither 0 nor 230 may intervene between the current character and the combining dot above.	After C:	([^\p{cc=230} \p{cc=0}])* [\u0307]
After_I	There is an uppercase I before C, and there is no intervening combining character class 230 (ABOVE) or 0.	Before C:	[I] ([^\p{cc=230} \p{cc=0}])*

In *Table 3-14*, a description of each context is followed by the equivalent regular expression(s) describing the context before C, the context after C, or both. The regular expressions use the syntax of Unicode Technical Standard #18, “Unicode Regular Expressions,” with one addition: “!” means that the expression does not match. All of the regular expressions are case-sensitive.

Default Case Conversion

The following specify the default case conversion operations for Unicode strings, in the absence of tailoring. In each instance, there are two variants: simple case conversion and full case conversion. In the full case conversion, the context-dependent mappings based on the casing context mentioned above must be used.

R1 ***toUppercase(X): Map each character C in X to default_upper(C).***

R2 ***toLowercase(X): Map each character C to default_lower(C).***

R3 ***toTitlecase(X): Find the word boundaries based on Unicode Standard Annex #29, “Text Boundaries.” Between each pair of word boundaries, find the first cased character F. If F exists, map F to default_title(F); then map each subsequent character C to default_lower(C).***

R4 ***toCasefold(X): Map each character C to default_fold(C).***

DRAFT

Default Case Detection

The specification of the default case detection of a string is based on the case conversion operations. Given a string `X`, and a string `Y = NFD(X)`, then:

- `isLowercase(X)` if and only if `toLowercase(Y) = Y`
- `isUppercase(X)` if and only if `toUppercase(Y) = Y`
- `isTitlecase(X)` if and only if `toTitlecase(Y) = Y`
- `isCasefolded(X)` if and only if `toCasefold(Y) = Y`
- `isCased(X)` if and only if ¬`isLowercase(Y)` or ¬`isUppercase(Y)` or ¬isTitlecase(Y)

The examples in *Table 3-15* show that these conditions are not mutually exclusive. "A2" is both uppercase and titlecase; "3" is uncased, so it is lowercase, uppercase, and titlecase.

Table 3-15. Case Detection Examples

Case	**Letter**	**Name**	**Alphanumeric**	**Digit**
Lowercase	a	john smith	a2	3
Uppercase	A	JOHN SMITH	A2	3
Titlecase	A	John Smith	A2	3

Default Caseless Matching

Default caseless (or case-insensitive) matching is specified by the following:

- A string `X` is a caseless match for a string `Y` if and only if:

```
toCasefold(X) = toCasefold(Y)
```

As described earlier, normally caseless matching should also use normalization, which means using one of the following operations:

- A string `X` is a canonical caseless match for a string `Y` if and only if:

```
NFD(toCasefold(NFD(X))) =
    NFD(toCasefold(NFD(Y)))
```

- A string `X` is a compatibility caseless match for a string `Y` if and only if:

```
NFKD(toCasefold(NFKD(toCasefold(NFD(X))))) =
    NFKD(toCasefold(NFKD(toCasefold(NFD(Y)))))
```

The invocations of normalization before folding in the above definitions are to catch very infrequent edge cases. Normalization is not required before folding, except for the character U+0345 ͅ COMBINING GREEK YPOGEGRAMMENI and any characters that have it as part of their decomposition, such as U+1FC3 ῃ GREEK SMALL LETTER ETA WITH YPOGEGRAMMENI.

In practice, optimized versions of implementations can catch these special cases and, thereby, avoid an extra normalization.

DRAFT

DRAFT