The Unicode® Standard
Version 10.0 – Core Specification

To learn about the latest version of the Unicode Standard, see http://www.unicode.org/versions/latest/.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

Unicode and the Unicode Logo are registered trademarks of Unicode, Inc., in the United States and other countries.

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

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

© 2017 Unicode, Inc.

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction. For information regarding permissions, inquire at http://www.unicode.org/reporting.html. For information about the Unicode terms of use, please see http://www.unicode.org/copyright.html.

The Unicode Standard / the Unicode Consortium; edited by the Unicode Consortium. — Version 10.0.
Includes bibliographical references and index.
ISBN 978-1-936213-16-0 (http://www.unicode.org/versions/Unicode10.0.0/)
1. Unicode (Computer character set) I. Unicode Consortium.
QA268.U545 2017

ISBN 978-1-936213-16-0
Published in Mountain View, CA
June 2017

Chapter 3

Conformance

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.

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, by contrast, 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. 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 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 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, their decompositions, and several other character properties for which stability is important to implementations. See also

"Stability of Properties" in *Section 3.5, Properties*. The formal statement of such stability guarantees is contained in the policies on character encoding stability found on the Unicode website. See the subsection "Policies" in *Section B.3, Other Unicode Online Resources*. See the discussion of backward compatibility in Section 2.5 of Unicode Standard Annex #31, "Unicode 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 5.2.0" indicates major version 5 of the Unicode Standard, minor version 2 of Unicode 5, and update version 0 of minor version Unicode 5.2.

Additional information on the current and past versions of the Unicode Standard can be found on the Unicode website. See the subsection "Versions" in *Section B.3, Other Unicode Online Resources*. 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.

Major and Minor Versions. Major and minor versions have significant additions to the standard, including, but not limited to, additions to the repertoire of encoded characters. Both are published as an updated core specification, together with associated updates to the code charts, the Unicode Standard Annexes and the Unicode Character Database. Such versions consolidate all errata and corrigenda and supersede any prior documentation for major, minor, or update versions.

A major version typically is of more importance to implementations; however, even update versions may be important to particular companies or other organizations. Major and minor versions are often synchronization points with related standards, such as with ISO/IEC 10646.

Prior to Version 5.2, minor versions of the standard were published as online amendments expressed as textual changes to the previous version, rather than as fully consolidated new editions of the core specification.

Update Version. An update version represents relatively small changes to the standard, typically updates to the data files of the Unicode Character Database. An update version never involves any additions to the character repertoire. These versions are published as modifications to the data files, and, on occasion, include documentation of small updates for selected errata or corrigenda.

Formally, each new version of the Unicode Standard supersedes all earlier versions. However, update versions generally do not obsolete the documentation of the immediately prior version of the standard.

Scheduling of Versions. Prior to Version 7.0.0, major, minor, and update versions of the Unicode Standard were published whenever the work on each new set of repertoire, prop-

erties, and documentation was finished. The emphasis was on ensuring synchronization of the major releases with corresponding major publication milestones for ISO/IEC 10646, but that practice resulted in an irregular publication schedule.

The Unicode Technical Committee changed its process as of Version 7.0.0 of the Unicode Standard, to make the publication time predictable. Major releases of the standard are now scheduled for annual publication. Further minor and update releases are not anticipated, but might occur under exceptional circumstances. This predictable, regular publication makes planning for new releases easier for most users of the standard. The detailed statements of synchronization between versions of the Unicode Standard and ISO/IEC 10646 have become somewhat more complex as a result, but in practice this has not been a problem for implementers.

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 website. See *Section B.3, Other Unicode Online Resources*, 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.3, Other Unicode Online Resources*. Whenever a new major or minor version of the standard is published, all errata up to that point are incorporated into the core specification, code charts, or other components of the standard.

Corrigenda. Occasionally errors may be 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.3, Other Unicode Online Resources*.

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 Unicode Character Database files can be found at Enumerated Version 3.1.1.

The reference for *this* version, Version 10.0.0, of the Unicode Standard, is

> The Unicode Consortium. The Unicode Standard, Version 10.0.0, defined by: *The Unicode Standard, Version 10.0* (Mountain View, CA: The Unicode Consortium, 2017. ISBN 978-1-936213-16-0)

References to an update (or minor version prior to Version 5.2.0) 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.3, Other Unicode Online Resources*.

Precision in Version Citation

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
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 4.

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 10.0.0 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 10.0.0 of the Unicode Standard.
>
> See Unicode Standard Annex #9, "Unicode Bidirectional Algorithm," (http://www.unicode.org/reports/tr9/tr9-37.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 website 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 the context that the Unicode Standard is the source of the specification.

Omitting a version number when referencing a Unicode algorithm may be appropriate when such a reference is meant as a generic reference to the overall algorithm. Such a generic reference may also be employed in the sense of latest available version of the algorithm. However, for specific and detailed conformance claims for Unicode algorithms, generic references are generally not sufficient, and a full version number must accompany the reference.

3.2 Conformance Requirements

This section presents the clauses specifying the formal conformance requirements for processes implementing Version 10.0 of the Unicode Standard.

In addition to this core specification, the Unicode Standard, Version 10.0.0, includes a number of Unicode Standard Annexes (UAXes) and the Unicode Character Database. At the end of this section there is a list of those annexes that are considered an integral part of the Unicode Standard, Version 10.0.0, 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.

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*.

Code Points Unassigned to Abstract Characters

C1 *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.

C2 *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.

C3 *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.
- Unassigned code points may have default property values. (See D26.)
- 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

Interpretation of characters is the key conformance requirement for the Unicode Standard, as it is for any coded character set standard. In legacy character set standards, the single conformance requirement is generally stated in terms of the interpretation of bit patterns used as characters. Conforming to a particular standard requires interpreting bit patterns used as characters according to the list of character names and the glyphs shown in the associated code table that form the bulk of that standard.

Interpretation of characters is a more complex issue for the Unicode Standard. It includes the core issue of interpreting code points used as characters according to the names and representative glyphs shown in the code charts, of course. However, the Unicode Standard also specifies character properties, behavior, and interactions between characters. Such information about characters is considered an integral part of the "character semantics established by this standard."

Information about the properties, behavior, and interactions between Unicode characters is provided in the Unicode Character Database and in the Unicode Standard Annexes. Additional information can be found throughout the other chapters of this core specification for the Unicode Standard. However, because of the need to keep extended discussions of scripts, sets of symbols, and other characters readable, material in other chapters is not always labeled as to its normative or informative status. In general, supplementary semantic information about a character is considered normative when it contributes directly to the identification of the character or its behavior. Additional information provided about the history of scripts, the languages which use particular characters, and so forth, is merely informative. Thus, for example, the rules about Devanagari rendering specified in *Section 12.1, Devanagari*, or the rules about Arabic character shaping specified in *Section 9.2, Arabic*, are normative: they spell out important details about how those characters behave in conjunction with each other that is necessary for proper and complete interpretation of the respective Unicode characters covered in each section.

C4 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.

C5 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.

C6 *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.
- 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 character sequences that are not part of the repertoire of a specified language (see *Section 5.12, Strategies for Handling Nonspacing Marks*).

Modification

C7 *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.*

- 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 replace the noncharacter with U+FFFD REPLACEMENT CHARACTER. If the implementation chooses to replace, delete or ignore a noncharacter, such an action constitutes a modification in the interpretation of the text. In general, a 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.
- Note that security problems can result if noncharacter code points are removed from text received from external sources. For more information, see *Section 23.7, Noncharacters*, and Unicode Technical Report #36, "Unicode Security Considerations."
- All processes and higher-level protocols are required to abide by conformance clause C7 at 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.
- There are important security issues associated with the correct interpretation and display of text. For more information, see Unicode Technical Report #36, "Unicode Security Considerations."

Character Encoding Forms

C8 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 D92.
- The specification of the code unit sequences for UTF-16 is given in D91.
- The specification of the code unit sequences for UTF-32 is given in D90.

C9 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 (D92) specifies that code unit sequences such as <C0 AF> are ill-formed.

C10 *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 D89.
- 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. There are important security issues associated with encoding conversion, especially with the conversion of malformed text. For more information, see Unicode Technical Report #36, "Unicode Security Considerations."

Character Encoding Schemes

C11 *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 are 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 D97.)

Bidirectional Text

C12 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.

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

Normalization Forms

C13 A process that produces Unicode text that purports to be in a Normalization Form shall do so in accordance with the specifications in Section 3.11, Normalization Forms.

C14 A process that tests Unicode text to determine whether it is in a Normalization Form shall do so in accordance with the specifications in Section 3.11, Normalization Forms.

C15 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

C16 Normative references to the Unicode 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.

C17 Higher-level protocols shall not make normative references to provisional properties.

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

Unicode Algorithms

C18 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 D17.
- 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.

C19 *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 Case 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.*

- 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 10.0 of the Unicode Standard. These annexes may contain either normative or informative material, or both. Any reference to Version 10.0 of the standard automatically includes these standard annexes.

- UAX #9: Unicode Bidirectional Algorithm, Version 10.0.0
- UAX #11: East Asian Width, Version 10.0.0
- UAX #14: Unicode Line Breaking Algorithm, Version 10.0.0
- UAX #15: Unicode Normalization Forms, Version 10.0.0
- UAX #24: Unicode Script Property, Version 10.0.0
- UAX #29: Unicode Text Segmentation, Version 10.0.0
- UAX #31: Unicode Identifier and Pattern Syntax, Version 10.0.0
- UAX #34: Unicode Named Character Sequences, Version 10.0.0
- UAX #38: Unicode Han Database (Unihan), Version 10.0.0
- UAX #41: Common References for Unicode Standard Annexes, Version 10.0.0
- UAX #42: Unicode Character Database in XML, Version 10.0.0
- UAX #44: Unicode Character Database, Version 10.0.0
- UAX #45: U-Source Ideographs, Version 10.0.0
- UAX #50: Unicode Vertical Text Layout, Version 10.0.0

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.

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:

- Character combination
- Canonical decomposition
- Compatibility decomposition
- Canonical ordering behavior
- Bidirectional behavior, as specified in the Unicode Bidirectional Algorithm (see Unicode Standard Annex #9, "Unicode Bidirectional Algorithm")
- Conjoining jamo behavior, as specified in *Section 3.12, Conjoining Jamo Behavior*
- Variation selection, as specified in *Section 23.4, Variation Selectors*
- Normalization, as specified in *Section 3.11, Normalization Forms*
- Default casing, as specified in *Section 3.13, Default Case Algorithms*

D2 *Character identity:* The identity of a character is established by its character name and representative glyph in the 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 assessed 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 24.1, 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.

D3 *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.

D4 *Character name:* A unique string used to identify each abstract character encoded in the standard.

- The character names in the Unicode Standard match those of the English edition of ISO/IEC 10646.
- Character names are immutable and cannot be overridden; they are stable identifiers. For more information, see *Section 4.8, Name*.
- The name of a Unicode character is also formally a character property in the Unicode Character Database. Its long property alias is "Name" and its short property alias is "na". Its value is the unique string label associated with the encoded character.
- The detailed specification of the Unicode character names, including rules for derivation of some ranges of characters, is given in *Section 4.8, Name*. That section also describes the relationship between the normative value of the Name property and the contents of the corresponding data field in UnicodeData.txt in the Unicode Character Database.

D5 *Character name alias:* An additional unique string identifier, other than the character name, associated with an encoded character in the standard.

- Character name aliases are assigned when there is a serious clerical defect with a character name, such that the character name itself may be misleading regarding the identity of the character. A character name alias constitutes an alternate identifier for the character.
- Character name aliases are also assigned to provide string identifiers for control codes and to recognize widely used alternative names and abbreviations for control codes, format characters and other special-use characters.
- Character name aliases are unique within the common namespace shared by character names, character name aliases, and named character sequences.
- More than one character name alias may be assigned to a given Unicode character. For example, the control code U+000D is given a character name alias for its ISO 6429 control function as CARRIAGE RETURN, but is also given a character name alias for its widely used abbreviation "CR".
- Character name aliases are a formal, normative part of the standard and should be distinguished from the informative, editorial aliases provided in the code

charts. See *Section 24.1, Character Names List*, for the notational conventions used to distinguish the two.

D6 *Namespace:* A set of names together with name matching rules, so that all names are distinct under the matching rules.

- Within a given namespace all names must be unique, although the same name may be used with a different meaning in a different namespace.
- Character names, character name aliases, and named character sequences share a single namespace in the Unicode Standard.

3.4 Characters and Encoding

D7 *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.

D8 *Abstract character sequence:* An ordered sequence of one or more abstract characters.

D9 *Unicode codespace:* A range of integers from 0 to $10FFFF_{16}$.

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

D10 *Code point:* Any value in the Unicode codespace.

- A code point is also known as a *code position*.
- See D77 for the definition of *code unit*.

D10a *Code point type:* Any of the seven fundamental classes of code points in the standard: Graphic, Format, Control, Private-Use, Surrogate, Noncharacter, Reserved.

- See *Table 2-3* for a summary of the meaning and use of each class.
- For Noncharacter, see also D14 Noncharacter.
- For Reserved, see also D15 Reserved code point.
- For Private-Use, see also D49 Private-use code point.
- For Surrogate, see also D71 High-surrogate code point and D73 Low-surrogate code point.

D10b *Block:* A named range of code points used to organize the allocation of characters.

- The exact list of blocks defined for each version of the Unicode Standard is specified by the data file Blocks.txt in the Unicode Character Database.

- The range for each defined block is specified by Field 0 in Blocks.txt; for example, "0000..007F".
- The ranges for blocks are non-overlapping. In other words, no code point can be contained in the range for one block and also in the range for a second distinct block.
- The range for each block is defined as a contiguous sequence. In other words, a block cannot consist of two (or more) discontiguous sequences of code points.
- Each range for a defined block starts with a value for which code point MOD 16 = 0 and terminates with a larger value for which code point MOD 16 = 15. This specification results in block ranges which always include full code point columns for code chart display. A block never starts or terminates in mid-column.
- All assigned characters are contained within ranges for defined blocks.
- Blocks may contain reserved code points, but no block contains *only* reserved code points. The majority of reserved code points are outside the ranges of defined blocks.
- A few designated code points are not contained within the ranges for defined blocks. This applies to the noncharacter code points at the last two code points of supplementary planes 1 through 14.
- The name for each defined block is specified by Field 1 in Blocks.txt; for example, "Basic Latin".
- The names for defined blocks constitute a unique namespace.
- The uniqueness rule for the block namespace is LM3, as defined in Unicode Standard Annex #44, "Unicode Character Database." In other words, casing, white space, hyphens, and underscores are ignored when matching strings for block names. The string "BASIC LATIN" or "Basic_Latin" would be considered as matching the name for the block named "Basic Latin".
- There is also a normative Block *property*. See *Table 3-2*. The Block property is a catalog property whose value is a string that identifies a block.
- Property value aliases for the Block property are defined in PropertyValueAliases.txt in the Unicode Character Database. The long alias defined for the Block property is always a loose match for the name of the block defined in Blocks.txt. Additional short aliases and other aliases are provided for convenience of use in regular expression syntax.
- The default value for the Block property is "No_Block". This default applies to any code point which is not contained in the range of a defined block.

For a general discussion of blocks and their relation to allocation in the Unicode Standard, see "Allocation Areas and Blocks" in *Section 2.8, Unicode Allocation*. For a general discus-

sion of the use of blocks in the presentation of the Unicode code charts, see *Chapter 24, About the Code Charts*.

D11 *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.

D12 *Coded character sequence:* An ordered sequence of one or more code points.

- A coded character sequence is also known as a *coded character representation*.
- Normally a coded character sequence consists of a sequence of encoded characters, but it may also include noncharacters or reserved code points.
- 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 conformance clause C2). Their removal by a conformant process constitutes modification of interpretation of the coded character sequence (see conformance clause C7).
- 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.

D13 *Deprecated character:* A coded character whose use is strongly discouraged.

- Deprecated characters are retained in the standard indefinitely, but should not be used. They are retained in the standard so that previously conforming data stay conformant in future versions of the standard.
- Deprecated characters typically consist of characters with significant architectural problems, or ones which cause implementation problems. Some examples

of characters deprecated on these grounds include tag characters (see *Section 23.9, Tag Characters*) and the alternate format characters (see *Section 23.3, Deprecated Format Characters*).

- Deprecated characters are explicitly indicated in the Unicode code charts. They are also given an explicit property value of Deprecated=True in the Unicode Character Database.
- 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.

D14 *Noncharacter:* A code point that is permanently reserved for internal use. 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 23.7, Noncharacters*.
- These code points are permanently reserved as noncharacters.

D15 *Reserved code point:* Any code point of the Unicode Standard that is reserved for future assignment. Also known as an *unassigned code point*.

- 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.

D16 *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.

D17 *Unicode algorithm:* The logical description of a process used to achieve a specified result involving Unicode characters.

- This definition, as 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.

D18 *Named Unicode algorithm:* A Unicode algorithm that is specified in the Unicode Standard or in other standards published by the Unicode Consortium and that is given an explicit name for ease of reference.

- Named Unicode algorithms are cited in titlecase in the Unicode Standard.

Table 3-1 lists the named Unicode algorithms and indicates the locations of their specifications. 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. When externally referenced, a named Unicode algorithm may be prefixed with the qualifier "Unicode" to make the connection of the algorithm to the Unicode Standard and other Unicode specifications clear. Thus, for example, the Bidirectional Algorithm is generally referred to by its full name, "Unicode Bidirectional Algorithm." As much as is practical, the titles of Unicode Standard Annexes which define Unicode algorithms consist of the name of the Unicode algorithm they specify. In a few cases, named Unicode algorithms are also widely known by their acronyms, and those acronyms are also listed in *Table 3-1*.

Table 3-1. Named Unicode Algorithms

Name	Description
Canonical Ordering	*Section 3.11*
Canonical Composition	*Section 3.11*
Normalization	*Section 3.11*
Hangul Syllable Composition	*Section 3.12*
Hangul Syllable Decomposition	*Section 3.12*
Hangul Syllable Name Generation	*Section 3.12*
Default Case Conversion	*Section 3.13*
Default Case Detection	*Section 3.13*
Default Caseless Matching	*Section 3.13*
Bidirectional Algorithm (UBA)	UAX #9
Line Breaking Algorithm	UAX #14
Character Segmentation	UAX #29
Word Segmentation	UAX #29
Sentence Segmentation	UAX #29
Hangul Syllable Boundary Determination	UAX #29
Standard Compression Scheme for Unicode (SCSU)	UTS #6
Unicode 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.

Types of Properties

D19 *Property:* A named attribute of an entity in the Unicode Standard, associated with a defined set of values.

- The lists of code point and encoded character properties for the Unicode Standard are documented in Unicode Standard Annex #44, "Unicode Character Database," and in Unicode Standard Annex #38, "Unicode Han Database (Unihan)."
- The file PropertyAliases.txt in the Unicode Character Database provides a machine-readable list of the non-Unihan properties and their names.

D20 *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.

D21 *Abstract character property:* A property of abstract characters.

- 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.

D22 *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 that are reserved for pattern syntax. (See Unicode Standard Annex #31, "Unicode 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, and Symbol) as well as code point properties (such as \p{gc=Cs} for the Surrogate code point property).

In the text of this standard the terms "Unicode character property," "character property," and "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

D23 *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".

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.

D24 *Explicit property value:* A value for an encoded character property that is explicitly associated with a code point in one of the data files of the Unicode Character Database.

D25 *Implicit property value:* A value for an encoded character property that is given by a 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 more than 1 million code points (most of them unassigned) for every property.

Default Property Values

To work properly in implementations, unassigned code points must be given default property values as if they were characters, because various algorithms require property values to be assigned to every code point before they can function at all.

Default property values are not uniform across all unassigned code points, because certain ranges of code points need different values for particular properties to maximize compatibility with expected future assignments. This means that some encoded character properties 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 as having the "AL" value, and the otherwise case as having the "L" value. For information on the default values for each encoded character property, see its description in the Unicode Character Database.

Default property values for unassigned code points are normative. They should not be changed by implementations to other values.

Default property values are also provided for private-use characters. Because the interpretation of private-use characters is subject to private agreement between the parties which exchange them, most default property values for those characters are overridable by higher-level protocols, to match the agreed-upon semantics for the characters. There are important exceptions for a few properties and Unicode algorithms. See *Section 23.5, Private-Use Characters*.

D26 *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. 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.

Classification of Properties by Their Values

D27 *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.

D28 *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.

- The General_Category and Bidi_Class properties are the only closed enumerations, except for the Boolean properties.

D29 *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.

D30 *Numeric property:* A numeric property is a property whose value is a number that can take on any integer or real value.

- 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.

D31 *String-valued property:* A property whose value is a string.

- The Canonical_Decomposition property is a string-valued property.

D32 *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 the Age, Block, and Script properties. Additional new values for the set of enumerated values for these properties may be added each time the standard is revised. A new value for Age is added for each new Unicode version,

a new value for Block is added for each new block added to the standard, and a new value for Script is added for each new script added to the standard.

Most properties have a single value associated with each code point. However, some properties may instead associate a set of multiple different values with each code point. See *Section 5.7.6, Properties Whose Values Are Sets of Values*, in Unicode Standard Annex #44, "Unicode Character Database."

Property Status

Each Unicode character property has one of several different statuses: normative, informative, contributory, or provisional. Each of these statuses is formally defined below, with some explanation and examples. In addition, normative properties can be subclassified, based on whether or not they can be overridden by conformant higher-level protocols.

The full list of currently defined Unicode character properties is provided in Unicode Standard Annex #44, "Unicode Character Database" and in Unicode Standard Annex #38, "Unicode Han Database (Unihan)." The tables of properties in those documents specify the status of each property explicitly. The data file PropertyAliases.txt provides a machine-readable listing of the character properties, except for those associated with the Unicode Han Database. The long alias for each property in PropertyAliases.txt also serves as the formal name of that property. In case of any discrepancy between the listing in PropertyAliases.txt and the listing in Unicode Standard Annex #44 or any other text of the Unicode Standard, the listing in PropertyAliases.txt should be taken as definitive. The tag for each Unihan-related character property documented in Unicode Standard Annex #38 serves as the formal name of that property.

D33 *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 Bidi_Class property 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. Nevertheless, they may require tailoring for particular local cultural conventions or particular implementations.

D34 *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.

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 each version of the standard.

Table 3-2. Normative Character Properties

Property	Description
Bidi_Class (directionality)	UAX #9 and *Section 4.4*
Bidi_Mirrored	UAX #9 and *Section 4.7*
Bidi_Paired_Bracket	UAX #9
Bidi_Paired_Bracket_Type	UAX #9
Block	*Section 24.1*
Canonical_Combining_Class	*Section 3.11* and *Section 4.3*
Case-related properties	*Section 3.13*, *Section 4.2*, and UAX #44
Composition_Exclusion	*Section 3.11*
Decomposition_Mapping	*Section 3.7* and *Section 3.11*
Default_Ignorable_Code_Point	*Section 5.21*
Deprecated	*Section 3.1*
General_Category	*Section 4.5*
Hangul_Syllable_Type	*Section 3.12* and UAX #29
Joining_Type and Joining_Group	*Section 9.2*
Name	*Section 4.8*
Noncharacter_Code_Point	*Section 23.7*
Numeric_Value	*Section 4.6*
White_Space	UAX #44

D35 *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 18.4* and UAX #11
Letter-related properties	*Section 4.10*
Line_Break	*Section 23.1*, *Section 23.2*, and UAX #14
Mathematical	*Section 22.5*
Script	UAX #24
Space	*Section 6.2* and *Table 6-2*
Unicode_1_Name	*Section 4.9*

D35a *Contributory property:* A simple property defined merely to make the statement of a rule defining a derived property more compact or general.

Contributory properties typically consist of short lists of exceptional characters which are used as part of the definition of a more generic normative or informative property. In most cases, such properties are given names starting with "Other", as Other_Alphabetic or Other_Default_Ignorable_Code_Point.

Contributory properties are not themselves subject to stability guarantees, but they are sometimes specified in order to make it easier to state the definition of a derived property which itself is subject to a stability guarantee, such as the derived, normative identifier-related properties, XID_Start and XID_Continue. The complete list of contributory properties is documented in Unicode Standard Annex #44, "Unicode Character Database."

D36 *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 data 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 database file (Unihan.zip) 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.

Section 4.12, Characters with Unusual Properties, provides additional lists of Unicode characters with unusual behavior, including many format controls discussed in detail elsewhere in the standard. Although in many instances those characters and their behavior have normative implications, the particular subclassification provided in *Table 4-10* does not directly correspond to any formal definition of Unicode character properties. Therefore that subclassification itself should also be considered provisional and potentially subject to change.

Context Dependence

D37 *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.

D38 *Context-independent property:* A property that is not context dependent; it applies to a code point in isolation.

Stability of Properties

D39 *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.

D40 *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.

- As new characters are assigned to previously unassigned code points, the replacement of any default values for these code points with actual property values must maintain stability.

D41 *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 include Age and 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 to which it was added, errors in the published listing of the property value could be corrected. For some other properties, even the correction of such errors is prohibited by explicit guarantees of property stability.

D42 *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* algorithms.
- 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. 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 nondefault values. Once that nondefault value is published, it can no longer be changed.

D43 *Stabilized property:* A property that is neither extended to new characters nor maintained in any other manner, but that is retained in the Unicode Character Database.

- A stabilized property is also a fixed property.

D44 *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 depre-

cated 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.

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.

The complete list of stability policies which affect character properties, their values, and their aliases, is available online. See the subsection "Policies" in *Section B.3, Other Unicode Online Resources*.

Simple and Derived Properties

D45 *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.

D46 *Derived property:* A Unicode character property whose values are algorithmically derived from some combination of simple properties.

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 include 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.

D47 *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. The long form serves as the formal name for the character property.
- Property aliases are defined in the file PropertyAliases.txt lists all of the non-Unihan properties that are part of each version of the standard. The Unihan properties are listed in Unicode Standard Annex #38, "Unicode Han Database (Unihan)."
- Property aliases of normative properties are themselves normative.

D48 *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 associated 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").

Private Use

D49 *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 are given default property values, but these default values are overridable by higher-level protocols that give those private-use code points a specific interpretation. See *Section 23.5, Private-Use Characters*.

3.6 Combination

Combining Character Sequences

D50 *Graphic character:* A character with the General Category 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), as well as the characters with the General Category 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*.

D51 *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 following 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.

D51a *Extended base:* Any base character, or any standard Korean syllable block.

- This term is defined to take into account the fact that sequences of Korean conjoining jamo characters behave as if they were a single Hangul syllable character, so that the entire sequence of jamos constitutes a base.
- For the definition of standard Korean syllable block, see D134 in *Section 3.12, Conjoining Jamo Behavior*.

D52 *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), Nonspacing 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 either 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—for example, 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.

D53 *Nonspacing mark:* A combining character with the General Category of Nonspacing Mark (Mn) or Enclosing Mark (Me).

- The position of a nonspacing mark in presentation depends 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.

D54 *Enclosing mark:* A nonspacing mark with the General Category of Enclosing Mark (Me).

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

D55 *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.
- Spacing marks such as U+0BCA TAMIL VOWEL SIGN O may be rendered on both sides of a base character, but are not enclosing marks.

D56 *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. 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>.

D56a *Extended combining character sequence:* A maximal character sequence consisting of either an extended base 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.

- Combining character sequence is commonly abbreviated as CCS, and extended combining character sequence is commonly abbreviated as ECCS.

D57 *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 D84.)

Grapheme Clusters

D58 *Grapheme base:* A character with the property Grapheme_Base, or any standard Korean syllable block.

- Characters with the property Grapheme_Base include all base characters (with the exception of U+FF9E..U+FF9F) 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. 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 D134 in *Section 3.12, Conjoining Jamo Behavior*.

D59 *Grapheme extender:* A character with the property Grapheme_Extend.

- Grapheme extender characters consist of all nonspacing marks, ZERO WIDTH JOINER, ZERO WIDTH NON-JOINER, U+FF9E, U+FF9F, and a small number of spacing marks.
- A grapheme extender can be conceived of primarily as the kind of nonspacing graphical mark that is 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 that have the property Grapheme_Extend are all the second parts of a two-part combining mark.
- The set of characters with the Grapheme_Extend property and the set of characters with the Grapheme_Base property are disjoint, by definition.

D60 *Grapheme cluster:* The text between grapheme cluster boundaries as specified by Unicode Standard Annex #29, "Unicode Text Segmentation."

- This definition of "grapheme cluster" is generic. The specification of grapheme cluster boundary segmentation in UAX #29 includes two alternatives, for "extended grapheme clusters" and for "legacy grapheme clusters." Furthermore, the segmentation algorithm in UAX #29 is tailorable.
- 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.
- 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 typically starts with a *grapheme base* and then extends across any subsequent sequence of *nonspacing* marks. A grapheme cluster is most directly relevant to text rendering and processes such as cursor placement and text selection in editing, but may also be relevant to comparison and searching.
- For many processes, a grapheme cluster behaves as if it were a single character with the same properties as its grapheme base. Effectively, nonspacing marks

apply graphically to the base, but do not change its properties. For example, <x, macron> behaves in line breaking or bidirectional layout as if it were the character x.

D61 *Extended grapheme cluster:* The text between extended grapheme cluster boundaries as specified by Unicode Standard Annex #29, "Unicode Text Segmentation."

- Extended grapheme clusters are defined in a parallel manner to legacy grapheme clusters, but also include sequences of *spacing* marks.
- Grapheme clusters and extended grapheme clusters may not have any particular linguistic significance, but are used to break up a string of text into units for processing.
- Grapheme clusters and extended grapheme clusters may be adjusted for particular processing requirements, by tailoring the rules for grapheme cluster segmentation specified in Unicode Standard Annex #29, "Unicode Text Segmentation."

Application of Combining Marks

A number of principles in the Unicode Standard relate to 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 characters 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 combining 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 so as 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*.

D61a *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.

D61b *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 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 12, South and Central Asia-I*, for many examples.) Similarly, sequences of base characters may form ligatures in rendering. (See "Cursive Connection and Ligatures" in *Section 23.2, Layout Controls*.)

The following listing specifies the principles regarding application of combining marks. Many of these principles are illustrated in *Section 2.11, Combining Characters*, and *Section 7.9, 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.
- 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". See *Figure 2-18*.

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. See *Figure 2-21*.
- 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. See *Table 2-6*.
- Side-by-side positioning may also reflect certain writing conventions, such as for titlo letters in the Old Church Slavonic manuscript tradition.
- When positioned side-by-side, the visual rendering order of a sequence of nonspacing marks reflects the dominant order of the script with which they are used. Thus, in Greek, the first nonspacing 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.
- The combining parentheses diacritical marks U+1ABB..U+1ABD are also positioned in a side-by-side manner, surrounding other diacritics, as described in the subsection "Combining Diacritical Marks Extended: U+1AB0–U+1AFF" in *Section 7.9, Combining Marks*.

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 *Figure 7-1*.

- Because of typographical conflict with the ascender of a base character, a combining háček (caron) is traditionally rendered as an apostrophe when placed, for example, on a lowercase "d". See *Figure 7-1*.
- 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 *Figure 9-5*.

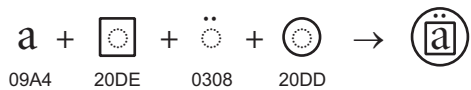
P5 ***[Normative] Nondistinct order. Nonspacing marks with different, non-zero combining classes may occur in different orders without affecting either the visual display of a combining character sequence or the interpretation of that sequence.***

- For example, if one nonspacing mark occurs above a grapheme base and another nonspacing mark occurs below it, they will have distinct combining classes. The order in which they occur in the combining character sequence does not matter for the display or interpretation of the resulting grapheme cluster.
- 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 clarify exactly which such alternate sequences must be considered as identical for display and interpretation. See *Figure 2-24*.
- 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 typed-in sequences and with canonically reordered sequences. See *Table 5-3*.
- 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 23.2, Layout Controls*.

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. 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.***

- Because such double diacritic display spans combinations of elements that would otherwise be considered grapheme clusters, the support of double diacritics in rendering may involve special handling for cursor placement and text selection. See *Figure 7-9* for an example.

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).***

- 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. See *Figure 7-10* for an example.
- 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. *Figure 7-11* shows an example of the use of CGJ to block this reordering.
- The interaction of enclosing marks and double diacritics is not well defined graphically. Many fonts and rendering processes may not be able to handle combinations of these marks. It is not recommended to use combinations of these together in the same grapheme cluster.

P9 ***[Guideline] When a nonspacing mark above—that is, a combining mark with ccc=230—is applied to the letters*** **i** ***and*** **j** ***or any other character with the Soft_Dotted property, the inherent dot on the base character is suppressed in display.***

- See *Figure 7-2* for an example.
- For languages such as Lithuanian, in which both a dot and an accent must be displayed, use U+0307 COMBINING DOT ABOVE. For guidelines in handling this situation in case mapping, see *Section 5.18, Case Mappings*.

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* → ㉮

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 23.2, Layout Controls*. For more discussion regarding the application of combining marks in general, see *Section 7.9, Combining Marks*.

3.7 Decomposition

D62 *Decomposition mapping:* A mapping from a character to a sequence of one or more characters that is a canonical or compatibility equivalent, and that is 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 24.1, Character Names List*).

D63 *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*.

- A decomposable character is also referred to as a *precomposed* character or *composite* character.
- The decomposition mappings from the Unicode Character Database are also given in *Section 24.1, Character Names List*.

D64 *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

D65 *Compatibility decomposition:* The decomposition of a character or character sequence 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, Normalization Forms*.

- The decomposition mappings from the Unicode Character Database are also given in *Section 24.1, Character Names List*.
- Some compatibility decompositions remove formatting information.

D66 *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 term should not be confused with the term "compatibility character," which is discussed in *Section 2.3, Compatibility Characters*.
- Many compatibility decomposable characters are included in the Unicode Standard solely 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.
- Some widely used and indispensable characters, such as NBSP, are compatibility decomposable characters for historical reasons. Their use is not discouraged.
- A large number of compatibility decomposable characters are used in phonetic and mathematical notation, where their use is not discouraged.
- For historical reasons, some characters that might have been given a compatibility decomposition were not, in fact, decomposed. The Normalization Stability Policy prohibits adding decompositions for such cases in the future, so that normalization forms will stay stable. See the subsection "Policies" in *Section B.3, Other Unicode Online Resources*.
- Replacing a compatibility decomposable character by its compatibility decomposition may lose round-trip convertibility with a base standard.

D67 *Compatibility equivalent:* Two character sequences are said to be compatibility equivalents if their full compatibility decompositions are identical.

Canonical Decomposition

D68 *Canonical decomposition:* The decomposition of a character or character sequence 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, Normalization Forms*.

- The decomposition mappings from the Unicode Character Database are also printed in *Section 24.1, Character Names List*.
- A canonical decomposition does not remove formatting information.

D69 *Canonical decomposable character:* A character that 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.

D70 *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* and is collated after *z*. In German, *ö* is weakly equivalent to *oe* and is collated with *oe*. In English, *ö* is just 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.

For information on the use of decomposition in normalization, see *Section 3.11, Normalization Forms*.

3.8 Surrogates

D71 *High-surrogate code point:* A Unicode code point in the range U+D800 to U+DBFF.

D72 *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.

D73 *Low-surrogate code point:* A Unicode code point in the range U+DC00 to U+DFFF.

D74 *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.

D75 *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 value 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 be used *only* 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 23.6, 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.

D76 *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.

D77 *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—that is, *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 "ANSI/ISO C wchar_t" in *Section 5.2, Programming Languages and Data Types*.

D78 *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*.
- 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 a 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.

D79 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 actually 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.
- 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.

D80 *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.

D81 *Unicode 8-bit string:* A Unicode string containing only UTF-8 code units.

D82 *Unicode 16-bit string:* A Unicode string containing only UTF-16 code units.

D83 *Unicode 32-bit string:* A Unicode string containing only UTF-32 code units.

D84 *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.

D84a *Ill-formed code unit subsequence:* A non-empty subsequence of a Unicode code unit sequence X which does not contain any code units which also belong to any minimal well-formed subsequence of X.

- In other words, an ill-formed code unit subsequence cannot overlap with a minimal well-formed subsequence.

D85 *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.

D85a *Minimal well-formed code unit subsequence:* A well-formed Unicode code unit sequence that maps to a single Unicode scalar value.

- For UTF-8, see the specification in D92 and *Table 3-7*.
- For UTF-16, see the specification in D91.
- For UTF-32, see the specification in D90.

A well-formed Unicode code unit sequence can be partitioned into one or more minimal well-formed code unit sequences for the given Unicode encoding form. Any Unicode code unit sequence can be partitioned into subsequences that are either well-formed or ill-formed. The sequence as a whole is well-formed if and only if it contains no ill-formed subsequence. The sequence as a whole is ill-formed if and only if it contains at least one ill-formed subsequence.

D86 *Well-formed UTF-8 code unit sequence:* A well-formed Unicode code unit sequence of UTF-8 code units.

- The UTF-8 code unit sequence <41 C3 B1 42> is well-formed, because it can be partitioned into subsequences, all of which match the specification for UTF-8 in *Table 3-7*. It consists of the following minimal well-formed code unit subsequences: <41>, <C3 B1>, and <42>.
- The UTF-8 code unit sequence <41 C2 C3 B1 42> is ill-formed, because it contains one ill-formed subsequence. There is no subsequence for the C2 byte which matches the specification for UTF-8 in *Table 3-7*. The code unit sequence

is partitioned into one minimal well-formed code unit subsequence, <41>, followed by one ill-formed code unit subsequence, <C2>, followed by two minimal well-formed code unit subsequences, <C3 B1> and <42>.

- In isolation, the UTF-8 code unit sequence <C2 C3> would be ill-formed, but in the context of the UTF-8 code unit sequence <41 C2 C3 B1 42>, <C2 C3> does not constitute an ill-formed code unit subsequence, because the C3 byte is actually the first byte of the minimal well-formed UTF-8 code unit subsequence <C3 B1>. Ill-formed code unit subsequences do not overlap with minimal well-formed code unit subsequences.

D87 *Well-formed UTF-16 code unit sequence:* A well-formed Unicode code unit sequence of UTF-16 code units.

D88 *Well-formed UTF-32 code unit sequence:* A well-formed Unicode code unit sequence of UTF-32 code units.

D89 *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.
- 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 a Unicode string *purports* to be *in* a Unicode encoding form, then it must not contain any ill-formed code unit subsequence.

If a process which verifies that a Unicode string is in a Unicode encoding form encounters an ill-formed code unit subsequence in that string, then it must not identify that string as being in that Unicode encoding form.

A process which interprets a Unicode string must not interpret any ill-formed code unit subsequences in the string as characters. (See conformance clause C10.) Furthermore, such a process must not treat any adjacent well-formed code unit sequences as being part of those ill-formed code unit sequences.

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

D90 *UTF-32 encoding form:* The Unicode encoding form that 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.
- 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*.

UTF-16

D91 *UTF-16 encoding form:* The Unicode encoding form that 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 that 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, UTF-8 and UTF-16*.

Table 3-5. UTF-16 Bit Distribution

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

Note: wwww = uuuuu - 1

UTF-8

D92 *UTF-8 encoding form:* The Unicode encoding form that assigns each Unicode scalar value to an unsigned byte sequence of one to four bytes in length, as specified in *Table 3-6* and *Table 3-7*.

- 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*.

- Because surrogate code points are not Unicode scalar values, any UTF-8 byte sequence that would otherwise map to code points U+D800..U+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, UTF-8 and UTF-16*.

Table 3-6. UTF-8 Bit Distribution

Scalar Value	First Byte	Second Byte	Third Byte	Fourth 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 "First 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	First Byte	Second Byte	Third Byte	Fourth 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

In *Table 3-7*, cases where a trailing byte range is not 80..BF are shown in bold italic to draw attention to them. These exceptions to the general pattern occur only in the second byte of a sequence.

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

D93 *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.

- 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. 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 C10.) 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.

Constraints on Conversion Processes

The requirement not to interpret any ill-formed code unit subsequences in a string as characters (see conformance clause C10) has important consequences for conversion processes. Such processes may, for example, interpret UTF-8 code unit sequences as Unicode character sequences. If the converter encounters an ill-formed UTF-8 code unit sequence which starts with a valid first byte, but which does not continue with valid successor bytes (see *Table 3-7*), it *must not* consume the successor bytes as part of the ill-formed subsequence whenever those successor bytes themselves constitute part of a well-formed UTF-8 code unit subsequence.

If an implementation of a UTF-8 conversion process stops at the first error encountered, without reporting the end of any ill-formed UTF-8 code unit subsequence, then the requirement makes little practical difference. However, the requirement does introduce a significant constraint if the UTF-8 converter continues past the point of a detected error, perhaps by substituting one or more U+FFFD replacement characters for the uninterpretable, ill-formed UTF-8 code unit subsequence. For example, with the input UTF-8 code unit sequence <C2 41 42>, such a UTF-8 conversion process must not return <U+FFFD> or <U+FFFD, U+0042>, because either of those outputs would be the result of misinterpreting a well-formed subsequence as being part of the ill-formed subsequence. The expected return value for such a process would instead be <U+FFFD, U+0041, U+0042>.

For a UTF-8 conversion process to consume valid successor bytes is not only non-conformant, but also leaves the converter open to security exploits. See Unicode Technical Report #36, "Unicode Security Considerations."

Although a UTF-8 conversion process is required to never consume well-formed subsequences as part of its error handling for ill-formed subsequences, such a process is not otherwise constrained in how it deals with any ill-formed subsequence itself. An ill-formed subsequence consisting of more than one code unit could be treated as a single error or as multiple errors. For example, in processing the UTF-8 code unit sequence <F0 80 80 41>, the only formal requirement mandated by Unicode conformance for a converter is that the <41> be processed and correctly interpreted as <U+0041>. The converter could return <U+FFFD, U+0041>, handling <F0 80 80> as a single error, or <U+FFFD, U+FFFD, U+FFFD, U+0041>, handling each byte of <F0 80 80> as a separate error, or could take other approaches to signalling <F0 80 80> as an ill-formed code unit subsequence.

Best Practices for Using U+FFFD. When using U+FFFD to replace ill-formed subsequences encountered during conversion, there are various logically possible approaches to associate U+FFFD with all or part of an ill-formed subsequence. To promote interoperability in the implementation of conversion processes, the Unicode Standard recommends a particular best practice. The following definitions simplify the discussion of this best practice:

D93a *Unconvertible offset:* An offset in a code unit sequence for which no code unit subsequence starting at that offset is well-formed.

D93b *Maximal subpart of an ill-formed subsequence:* The longest code unit subsequence starting at an unconvertible offset that is either:

a. the initial subsequence of a well-formed code unit sequence, or

b. a subsequence of length one.

- The term *maximal subpart of an ill-formed subsequence* can be abbreviated to *maximal subpart* when it is clear in context that the subsequence in question is ill-formed.
- This definition can be trivially applied to the UTF-32 or UTF-16 encoding forms, but is primarily of interest when converting UTF-8 strings.
- For example, in the ill-formed UTF-8 sequence <41 C0 AF 41 F4 80 80 41>, there are two ill-formed subsequences: <C0 AF> and <F4 80 80>, each separated by <41>, which is well-formed. Applying the definition of maximal subparts for these ill-formed subsequences, in the first case <C0> is a maximal subpart, because that byte value can never be the first byte of a well-formed UTF-8 sequence. In the second subsequence, <F4 80 80> is a maximal subpart, because up to that point all three bytes match the specification for UTF-8. It is only when followed by <41> that the sequence of <F4 80 80> can be determined to be ill-formed, because the specification requires a following byte in the range 80..BF, instead.

- Another example illustrates the application of the concept of maximal subpart for UTF-8 continuation bytes outside the allowable ranges defined in *Table 3-7*. The UTF-8 sequence <41 E0 9F 80 41> is ill-formed, because <9F> is not an allowed second byte of a UTF-8 sequence commencing with <E0>. In this case, there is an unconvertible offset at <E0> and the maximal subpart at that offset is also <E0>. The subsequence <E0 9F> cannot be a maximal subpart, because it is not an initial subsequence of any well-formed UTF-8 code unit sequence.

Using the definition for maximal subpart, the best practice can be stated simply as:

> ***Whenever an unconvertible offset is reached during conversion of a code unit sequence:***
>
> ***1. The maximal subpart at that offset should be replaced by a single U+FFFD.***
>
> ***2. The conversion should proceed at the offset immediately after the maximal subpart.***

This sounds complicated, but it reflects the way optimized conversion processes are typically constructed, particularly for UTF-8. A sequence of code units will be processed up to the point where the sequence either can be unambiguously interpreted as a particular Unicode code point or where the converter recognizes that the code units collected so far constitute an ill-formed subsequence. At that point, the converter can emit a single U+FFFD for the collected (but ill-formed) code unit(s) and move on, without having to further accumulate state. The maximal subpart could be the start of a well-formed sequence, except that the sequence lacks the proper continuation. Alternatively, the converter may have found a continuation code unit or some other code unit which cannot be the start of a well-formed sequence.

To illustrate this policy, consider the ill-formed UTF-8 sequence <61 F1 80 80 E1 80 C2 62 80 63 80 BF 64>. Possible alternative approaches for a UTF-8 converter using U+FFFD are illustrated in *Table 3-8*.

Table 3-8. Use of U+FFFD in UTF-8 Conversion

	61	F1	80	80	E1	80	C2	62	80	63	80	BF	64
1	0061	FFFD						0062	FFFD	0063	FFFD		0064
2	0061	FFFD			FFFD		FFFD	0062	FFFD	0063	FFFD	FFFD	0064
3	0061	FFFD	FFFD	FFFD	FFFD	FFFD	FFFD	0062	FFFD	0063	FFFD	FFFD	0064

The recommended conversion policy would have the outcome shown in Row 2 of *Table 3-8*, rather than Row 1 or Row 3. For example, a UTF-8 converter would detect that <F1 80 80> constituted a maximal subpart of the ill-formed subsequence as soon as it encountered the subsequent code unit <E1>, so at that point, it would emit a single U+FFFD and then continue attempting to convert from the <E1> code unit—and so forth to the end of the code unit sequence to convert. The UTF-8 converter would detect that the code unit <80> in the sequence <62 80 63> is not well-formed, and would replace it by

U+FFFD. Neither of the code units <80> or <BF> in the sequence <63 80 BF 64> is the start of a potentially well-formed sequence; therefore *each* of them is separately replaced by U+FFFD. For a discussion of the generalization of this approach for conversion of other character sets to Unicode, see *Section 5.22, Best Practice for U+FFFD Substitution*.

3.10 Unicode Encoding Schemes

D94 *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.

D95 *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 a near-certain indication that the data stream is using the UTF-8 encoding scheme.

D96 *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.

D97 *UTF-16LE encoding scheme:* The Unicode encoding scheme that serializes a UTF-16 code unit sequence as a byte sequence in little-endian format.

- 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.

D98 *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; 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 3-9 gives examples that summarize the three Unicode encoding schemes for the UTF-16 encoding form.

Table 3-9. 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

D99 *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.

D100 *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.

D101 *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; 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 3-10 gives examples that summarize the three Unicode encoding schemes for the UTF-32 encoding form.

Table 3-10. 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

Table 3-10. Summary of UTF-32BE, UTF-32LE, and UTF-32 (Continued)

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

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 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*. These terms 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, thereby 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 behavior according to conformance clause C7, 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 the initial byte sequence <EF BB BF> as a signature on UTF-8 byte sequences is not recommended by the Unicode Standard.

3.11 Normalization Forms

The concepts of canonical equivalent (D70) or compatibility equivalent (D67) characters in the Unicode Standard make it necessary to have a full, formal definition of equivalence for Unicode strings. String equivalence is determined by a process called *normalization*, whereby strings are converted into forms which are compared directly for identity.

This section provides the formal definitions of the four Unicode Normalization Forms. It defines the Canonical Ordering Algorithm and the Canonical Composition Algorithm which are used to convert Unicode strings to one of the Unicode Normalization Forms for comparison. It also formally defines Unicode Combining Classes—values assigned to all Unicode characters and used by the Canonical Ordering Algorithm.

Note: In versions of the Unicode Standard up to Version 5.1.0, the Unicode Normalization Forms and the Canonical Composition Algorithm were defined in Unicode Standard Annex #15, "Unicode Normalization Forms." Those definitions have now been consolidated in this chapter, for clarity of exposition of the normative definitions and algorithms involved in Unicode normalization. However, because implementation of Unicode normalization is quite complex, implementers are still advised to fully consult Unicode Standard Annex #15, "Unicode Normalization Forms," which contains more detailed explanations, examples, and implementation strategies.

Unicode normalization should be carefully distinguished from Unicode collation. Both processes involve comparison of Unicode strings. However, the point of Unicode normalization is to make a determination of canonical (or compatibility) equivalence or non-equivalence of strings—it does not provide any rank-ordering information about those strings. Unicode collation, on the other hand, is designed to provide orderable weights or "keys" for strings; those keys can then be used to sort strings into ordered lists. Unicode normalization is not tailorable; normalization equivalence relationships between strings are exact and unchangeable. Unicode collation, on the other hand, is designed to be tailorable to allow many kinds of localized and other specialized orderings of strings. For more information, see Unicode Technical Standard #10, "Unicode Collation Algorithm."

D102 [Moved to *Section 3.6, Combination* and renumbered as D61a.]

D103 [Moved to *Section 3.6, Combination* and renumbered as D61b.]

Normalization Stability

A very important attribute of the Unicode Normalization Forms is that they must remain stable between versions of the Unicode Standard. A Unicode string normalized to a particular Unicode Normalization Form in one version of the standard is guaranteed to remain in that Normalization Form for implementations of future versions of the standard. In order to ensure this stability, there are strong constraints on changes of any character properties that are involved in the specification of normalization—in particular, the combining class and the decomposition of characters. The details of those constraints are spelled out in the Normalization Stability Policy. See the subsection "Policies" in *Section B.3, Other*

Unicode Online Resources. The requirement for stability of normalization also constrains what kinds of characters can be encoded in future versions of the standard. For an extended discussion of this topic, see *Section 3, Versioning and Stability*, in Unicode Standard Annex #15, "Unicode Normalization Forms."

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 used to determine whether two character sequences are considered identical for interpretation.

D104 *Combining class:* A numeric value in the range 0..254 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 \p{Canonical_Combining_Class = 0} (or \p{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, with a few exceptions. Also, not all nonspacing marks have a non-zero combining class. Thus, while the correlation between ^\p{ccc=0] and \p{gc=Mn} is close, it is not exact, and implementations should not depend on the two concepts being identical.

D105 *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 positions were conceived of as occurring in a fixed position with respect to their grapheme base, regardless of any other combining mark that might also apply to the 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.

D106 *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 placed above the base character are given the same value, \p{ccc=230}, while all generic nonspacing marks placed below are given the value \p{ccc=220}. Nonspacing marks that tend to sit on one "shoulder" or another of a grapheme base, or that may actually be attached to the grapheme base itself when applied, have their own combining classes.

The design of canonical ordering generally assures that:

- When two combining characters C1 and C2 *do* typographically interact, the sequence C1+ C2 is *not* canonically equivalent to C2+ C1.
- When two combining characters C1 and C2 do *not* typographically interact, the sequence C1+ C2 *is* canonically equivalent to C2+ C1.

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 involving 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 be taken only 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 that are not.

Specification of Unicode Normalization Forms

The specification of Unicode Normalization Forms applies to all Unicode coded character sequences (D12). For clarity of exposition in the definitions and rules specified here, the terms "character" and "character sequence" are used, but coded character sequences refer also to sequences containing noncharacters or reserved code points. Unicode Normalization Forms are specified for *all* Unicode code points, and not just for ordinary, assigned graphic characters.

Starters

D107 *Starter:* Any code point (assigned or not) with combining class of zero (ccc=0).

- Note that ccc=0 is the default value for the Canonical_Combining_Class property, so that all reserved code points are Starters by definition. Noncharacters are also Starters by definition. All control characters, format characters, and private-use characters are also Starters.

- Private agreements cannot override the value of the Canonical_Combining_Class property for private-use characters.

Among the graphic characters, all those with General_Category values other than gc=M are Starters. Some combining marks have ccc=0 and thus are also Starters. Combining marks with ccc other than 0 are not Starters. *Table 3-11* summarizes the relationship between types of combining marks and their status as Starters.

Table 3-11. Combining Marks and Starter Status

Description	gc	ccc	Starter
Nonspacing	Mn	0	Yes
		>0	No
Spacing	Mc	0	Yes
		>0	No
Enclosing	Me	0	Yes

The term *Starter* refers, in concept, to the starting character of a combining character sequence (D56), because all combining character sequences except defective combining character sequences (D57) commence with a ccc=0 character—in other words, they start with a Starter. However, because the specification of Unicode Normalization Forms must apply to all possible coded character sequences, and not just to typical combining character sequences, the behavior of a code point for Unicode Normalization Forms is specified entirely in terms of its status as a Starter or a non-starter, together with its Decomposition_Mapping value.

Canonical Ordering Algorithm

D108 *Reorderable pair:* Two adjacent characters A and B in a coded character sequence <A, B> are a Reorderable Pair if and only if ccc(A) > ccc(B) > 0.

D109 *Canonical Ordering Algorithm:* In a decomposed character sequence D, exchange the positions of the characters in each Reorderable Pair until the sequence contains no more Reorderable Pairs.

- In effect, the Canonical Ordering Algorithm is a local bubble sort that guarantees that a Canonical Decomposition or a Compatibility Decomposition will contain no subsequences in which a combining mark is *followed* directly by another combining mark that has a lower, non-zero combining class.
- Canonical ordering is defined in terms of application of the Canonical Ordering Algorithm to an *entire* decomposed sequence. For example, canonical decomposition of the sequence <U+1E0B LATIN SMALL LETTER D WITH DOT ABOVE, U+0323 COMBINING DOT BELOW> would result in the sequence <U+0064 LATIN SMALL LETTER D, U+0307 COMBINING DOT ABOVE, U+0323 COMBINING DOT BELOW>, a sequence which is not yet in canonical order. Most decompositions for Unicode strings are already in canonical order.

Table 3-12 gives some examples of sequences of characters, showing which of them constitute a Reorderable Pair and the reasons for that determination. Except for the base character "a", the other characters in the example table are combining marks; character names are abbreviated in the Sequence column to make the examples clearer.

Table 3-12. Reorderable Pairs

Sequence	Combining Classes	Reorderable?	Reason
<a, acute>	0, 230	No	ccc(A)=0
<acute, a>	230, 0	No	ccc(B)=0
<diaeresis, acute>	230, 230	No	ccc(A)=ccc(B)
<cedilla, acute>	202, 230	No	ccc(A)<ccc(B)
<acute, cedilla>	230, 202	Yes	ccc(A)>ccc(B)

Canonical Composition Algorithm

D110 *Singleton decomposition:* A canonical decomposition mapping from a character to a different single character.

- The default value for the Decomposition_Mapping property for a code point (including any private-use character, any noncharacter, and any unassigned code point) is the code point itself. This default value does not count as a singleton decomposition, because it does not map a character to a *different* character. Private agreements cannot override the decomposition mapping for private-use characters
- Example: U+2126 OHM SIGN has a singleton decomposition to U+03A9 GREEK CAPITAL LETTER OMEGA.
- A character with a singleton decomposition is often referred to simply as a *singleton* for short.

D110a *Expanding canonical decomposition*: A canonical decomposition mapping from a character to a sequence of more than one character.

D110b *Starter decomposition*: An expanding canonical decomposition for which both the character being mapped and the first character of the resulting sequence are Starters.

- Definitions D110a and D110b are introduced to simplify the following definition of non-starter decomposition and make it more precise.

D111 *Non-starter decomposition:* An expanding canonical decomposition which is not a starter decomposition.

- Example: U+0344 COMBINING GREEK DIALYTIKA TONOS has an expanding canonical decomposition to the sequence <U+0308 COMBINING DIAERESIS, U+0301 COMBINING ACUTE ACCENT>. U+0344 is a non-starter, and the first

character in its decomposition is a non-starter. Therefore, on two counts, U+0344 has a non-starter decomposition.

- Example: U+0F73 TIBETAN VOWEL SIGN II has an expanding canonical decomposition to the sequence <U+0F71 TIBETAN VOWEL SIGN AA, U+0F72 TIBETAN VOWEL SIGN I>. The first character in that sequence is a non-starter. Therefore U+0F73 has a non-starter decomposition, even though U+0F73 is a Starter.
- As of the current version of the standard, there are no instances of the third possible situation: a non-starter character with an expanding canonical decomposition to a sequence whose first character is a Starter.

D112 Composition exclusion: A Canonical Decomposable Character (D69) which has the property value Composition_Exclusion=True.

- The list of Composition Exclusions is provided in CompositionExclusions.txt in the Unicode Character Database.

D113 Full composition exclusion: A Canonical Decomposable Character which has the property value Full_Composition_Exclusion=True.

- Full composition exclusions consist of the entire list of composition exclusions plus all characters with singleton decompositions or with non-starter decompositions.
- For convenience in implementation of Unicode normalization, the derived property Full_Composition_Exclusion is computed, and all characters with the property value Full_Composition_Exclusion=True are listed in DerivedNormalizationProps.txt in the Unicode Character Database.

D114 Primary composite: A Canonical Decomposable Character (D69) which is not a Full Composition Exclusion.

- For any given version of the Unicode Standard, the list of primary composites can be computed by extracting all canonical decomposable characters from UnicodeData.txt in the Unicode Character Database, adding the list of precomposed Hangul syllables (D132), and subtracting the list of Full Decomposition Exclusions.

D115 Blocked: Let A and C be two characters in a coded character sequence <A, ... C>. C is blocked from A if and only if ccc(A)=0 and there exists some character B between A and C in the coded character sequence, i.e., <A, ... B, ... C>, and either ccc(B)=0 or ccc(B) >= ccc(C).

- Because the Canonical Composition Algorithm operates on a string which is already in canonical order, testing whether a character is blocked requires looking only at the immediately preceding character in the string.

D116 Non-blocked pair: A pair of characters <A, ... C> in a coded character sequence, in which C is not blocked from A.

- It is important for proper implementation of the Canonical Composition Algorithm to be aware that a Non-blocked Pair need not be contiguous.

D117 *Canonical Composition Algorithm:* Starting from the second character in the coded character sequence (of a Canonical Decomposition or Compatibility Decomposition) and proceeding sequentially to the final character, perform the following steps:

R1 ***Seek back (left) in the coded character sequence from the character C to find the last Starter L preceding C in the character sequence.***

R2 ***If there is such an L, and C is not blocked from L, and there exists a Primary Composite P which is canonically equivalent to the sequence <L, C>, then replace L by P in the sequence and delete C from the sequence.***

- When the algorithm completes, all Non-blocked Pairs canonically equivalent to a Primary Composite will have been systematically replaced by those Primary Composites.
- The replacement of the Starter L in ***R2*** requires continuing to check the succeeding characters until the character at that position is no longer part of any Non-blocked Pair that can be replaced by a Primary Composite. For example, consider the following hypothetical coded character sequence: <U+007A *z*, U+0335 *short stroke overlay*, U+0327 *cedilla*, U+0324 *diaeresis below*, U+0301 *acute*>. None of the first three combining marks forms a Primary Composite with the letter *z*. However, the fourth combining mark in the sequence, *acute*, does form a Primary Composite with *z*, and it is not Blocked from the *z*. Therefore, ***R2*** mandates the replacement of the sequence <U+007A *z*, ... U+0301 *acute*> with <U+017A *z-acute*, ...>, even though there are three other combining marks intervening in the sequence.
- The character C in ***R1*** is not necessarily a non-starter. It is necessary to check *all* characters in the sequence, because there are sequences <L, C> where *both* L and C are Starters, yet there is a Primary Composite P which is canonically equivalent to that sequence. For example, Indic two-part vowels often have canonical decompositions into sequences of two spacing vowel signs, each of which has Canonical_Combining_Class=0 and which is thus a Starter by definition. Nevertheless, such a decomposed sequence has an equivalent Primary Composite.

Definition of Normalization Forms

The Unicode Standard specifies four normalization forms. Informally, two of these forms are defined by maximal *decomposition* of equivalent sequences, and two of these forms are defined by maximal *composition* of equivalent sequences. Each is then differentiated based on whether it employs a Canonical Decomposition or a Compatibility Decomposition.

D118 *Normalization Form D (NFD):* The Canonical Decomposition of a coded character sequence.

D119 *Normalization Form KD (NFKD):* The Compatibility Decomposition of a coded character sequence.

D120 *Normalization Form C (NFC):* The Canonical Composition of the Canonical Decomposition of a coded character sequence.

D121 *Normalization Form KC (NFKC):* The Canonical Composition of the Compatibility Decomposition of a coded character sequence.

Logically, to get the NFD or NFKD (maximally decomposed) normalization form for a Unicode string, one first computes the full decomposition of that string and then applies the Canonical Ordering Algorithm to it.

Logically, to get the NFC or NFKC (maximally composed) normalization form for a Unicode string, one first computes the NFD or NFKD normalization form for that string, and then applies the Canonical Composition Algorithm to it.

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 canonical decomposition of precomposed Hangul syllables.
- Compose jamo characters into precomposed Hangul syllables.
- Algorithmically determine the names of precomposed Hangul syllables.

For more information, see the "Hangul Syllables" and "Hangul Jamo" subsections in *Section 18.6, Hangul*. Hangul syllables are a special case of grapheme clusters. For the algorithm to determine syllable boundaries in a sequence of conjoining jamo characters, see Section 8, "Hangul Syllable Boundary Determination" in Unicode Standard Annex #29, "Unicode Text Segmentation."

Definitions

The following definitions use the Hangul_Syllable_Type property, which is defined in the UCD file HangulSyllableType.txt.

D122 *Leading consonant:* A character with the Hangul_Syllable_Type property value Leading_Jamo. Abbreviated as L.

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

D123 *Choseong:* A sequence of one or more leading consonants.

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

D124 *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.

D125 *Vowel:* A character with the Hangul_Syllable_Type property value Vowel_Jamo. Abbreviated as V.

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

D126 *Jungseong:* A sequence of one or more vowels.

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

D127 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.

D128 Trailing consonant: A character with the Hangul_Syllable_Type property value Trailing_Jamo. Abbreviated as T.

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

D129 Jongseong: A sequence of one or more trailing consonants.

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

D130 LV_Syllable: A character with Hangul_Syllable_Type property value LV_Syllable. Abbreviated as LV.

- An LV_Syllable has a canonical decomposition to a sequence of the form <L, V>.

D131 LVT_Syllable: A character with Hangul_Syllable_Type property value LVT_Syllable. Abbreviated as LVT.

- An LVT_Syllable has a canonical decomposition to a sequence of the form <LV, T>.

D132 Precomposed Hangul syllable: A character that is either an LV_Syllable or an LVT_Syllable.

D133 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.

D134 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, or any other sequence that is canonically equivalent.

- All precomposed Hangul syllables, which have the form LV or LVT, are standard Korean syllable blocks.
- Alternatively, a standard Korean syllable block may be expressed as a sequence of a choseong and a jungseong, optionally followed by a jongseong.
- A choseong filler may substitute for a missing leading consonant, and a jungseong filler may substitute for a missing vowel.

- This definition is used in Unicode Standard Annex #29, "Unicode Text Segmentation," as part of the algorithm for determining syllable boundaries in a sequence of conjoining jamo characters.

Hangul Syllable Decomposition

The following algorithm specifies how to take a precomposed Hangul syllable *s* and arithmetically derive its full canonical decomposition *d*. This normative mapping for precomposed Hangul syllables is referenced by D68, Canonical decomposition, in *Section 3.7, Decomposition*.

This algorithm, as well as the other Hangul-related algorithms defined in the following text, is first specified in pseudo-code. Then each is exemplified, showing its application to a particular Hangul character or sequence. The Hangul characters used in those examples are shown in *Table 3-13*. Finally, each algorithm is then further exemplified with an implementation as a Java method at the end of this section.

Table 3-13. Hangul Characters Used in Examples

Code Point	Glyph	Character Name	Jamo Short Name
U+D4DB	퓛	HANGUL SYLLABLE PWILH	
U+1111	ㅍ	HANGUL CHOSEONG PHIEUPH	P
U+1171	ㅟ	HANGUL JUNGSEONG WI	WI
U+11B6	ᶇ	HANGUL JONGSEONG RIEUL-HIEUH	LH

Common Constants. Define the following constants:

```
SBase = AC00₁₆
LBase = 1100₁₆
VBase = 1161₁₆
TBase = 11A7₁₆
LCount = 19
VCount = 21
TCount = 28
NCount = 588 (VCount * TCount)
SCount = 11172 (LCount * NCount)
```

TBase is set to one less than the beginning of the range of trailing consonants, which starts at U+11A8. TCount is set to one more than the number of trailing consonants relevant to the decomposition algorithm: ($11C2_{16}$ - $11A8_{16}$ + 1) + 1. NCount is thus the number of precomposed Hangul syllables starting with the same leading consonant, counting both the LV_Syllables and the LVT_Syllables for each possible trailing consonant. SCount is the total number of precomposed Hangul syllables.

Syllable Index. First compute the index of the precomposed Hangul syllable *s*:

```
SIndex = s - SBase
```

Arithmetic Decomposition Mapping. If the precomposed Hangul syllable *s* with the index SIndex (defined above) has the Hangul_Syllable_Type value LV, then it has a canonical decomposition mapping into a sequence of an L jamo and a V jamo, <LPart, VPart>:

```
LIndex = SIndex div NCount
VIndex = (SIndex mod NCount) div TCount

LPart = LBase + LIndex
VPart = VBase + VIndex
```

If the precomposed Hangul syllable *s* with the index SIndex (defined above) has the Hangul_Syllable_Type value LVT, then it has a canonical decomposition mapping into a sequence of an LV_Syllable and a T jamo, <LVPart, TPart>:

```
LVIndex = (SIndex div TCount) * TCount
TIndex = SIndex mod TCount

LVPart = SBase + LVIndex
TPart = TBase + TIndex
```

In this specification, the "div" operator refers to integer division (rounded down). The "mod" operator refers to the modulo operation, equivalent to the integer remainder for positive numbers.

The canonical decomposition mappings calculated this way are equivalent to the values of the Unicode character property Decomposition_Mapping (dm), for each precomposed Hangul syllable.

Full Canonical Decomposition. The full canonical decomposition for a Unicode character is defined as the recursive application of canonical decomposition mappings. The canonical decomposition mapping of an LVT_Syllable contains an LVPart which itself is a precomposed Hangul syllable and thus must be further decomposed. However, it is simplest to unwind the recursion and directly calculate the resulting <LPart, VPart, TPart> sequence instead. For full canonical decomposition of a precomposed Hangul syllable, compute the indices and components as follows:

```
LIndex = SIndex div NCount
VIndex = (SIndex mod NCount) div TCount
TIndex = SIndex mod TCount

LPart = LBase + LIndex
VPart = VBase + VIndex
TPart = TBase + TIndex  if TIndex > 0
```

If TIndex = 0, then there is no trailing consonant, so map the precomposed Hangul syllable *s* to its full decomposition *d* = <LPart, VPart>. Otherwise, there is a trailing consonant, so map *s* to its full decomposition *d* = <LPart, VPart, TPart>.

Example. For the precomposed Hangul syllable U+D4DB, compute the indices and components:

```
SIndex = 10459
LIndex = 17
VIndex = 16
```

```
TIndex = 15
LPart = LBase + 17 = 1111₁₆
VPart = VBase + 16 = 1171₁₆
TPart = TBase + 15 = 11B6₁₆
```

Then map the precomposed syllable to the calculated sequence of components, which constitute its full canonical decomposition:

```
U+D4DB → <U+1111, U+1171, U+11B6>
```

Note that the canonical decomposition mapping for U+D4DB would be <U+D4CC, U+11B6>, but in computing the full canonical decomposition, that sequence would only be an intermediate step.

Hangul Syllable Composition

The following algorithm specifies how to take a canonically decomposed sequence of Hangul jamo characters *d* and arithmetically derive its mapping to an equivalent precomposed Hangul syllable *s*. This normative mapping can be used to calculate the Primary Composite for a sequence of Hangul jamo characters, as specified in D117, Canonical Composition Algorithm, in *Section 3.11, Normalization Forms*. Strictly speaking, this algorithm is simply the inverse of the full canonical decomposition mappings specified by the Hangul Syllable Decomposition Algorithm. However, it is useful to have a summary specification of that inverse mapping as a separate algorithm, for convenience in implementation.

Note that the presence of any non-jamo starter or any combining character between two of the jamos in the sequence *d* would constitute a blocking context, and would prevent canonical composition. See D115, Blocked, in *Section 3.11, Normalization Forms*.

Arithmetic Primary Composite Mapping. Given a Hangul jamo sequence <LPart, VPart>, where the LPart is in the range U+1100..U+1112, and where the VPart is in the range U+1161..U+1175, compute the indices and syllable mapping:

```
LIndex = LPart - LBase
VIndex = VPart - VBase
LVIndex = LIndex * NCount + VIndex * TCount
s = SBase + LVIndex
```

Given a Hangul jamo sequence <LPart, VPart, TPart>, where the LPart is in the range U+1100..U+1112, where the VPart is in the range U+1161..U+1175, and where the TPart is in the range U+11A8..U+11C2, compute the indices and syllable mapping:

```
LIndex = LPart - LBase
VIndex = VPart - VBase
TIndex = TPart - TBase
LVIndex = LIndex * NCount + VIndex * TCount
s = SBase + LVIndex + TIndex
```

The mappings just specified deal with canonically decomposed sequences of Hangul jamo characters. However, for completeness, the following mapping is also defined to deal with

cases in which Hangul data is not canonically decomposed. Given a sequence <LVPart, TPart>, where the LVPart is a precomposed Hangul syllable of Hangul_Syllable_Type LV, and where the TPart is in the range U+11A8..U+11C2, compute the index and syllable mapping:

```
TIndex = TPart - TBase
s = LVPart + TIndex
```

Example. For the canonically decomposed Hangul jamo sequence <U+1111, U+1171, U+11B6>, compute the indices and syllable mapping:

```
LIndex = 17
VIndex = 16
TIndex = 15
LVIndex = 17 * 588 + 16 * 28 = 9996 + 448 = 10444
s = AC00_16 + 10444 + 15 = D4DB_16
```

Then map the Hangul jamo sequence to this precomposed Hangul syllable as its Primary Composite:

```
<U+1111, U+1171, U+11B6> → U+D4DB
```

Hangul Syllable Name Generation

The Unicode character names for precomposed Hangul syllables are derived algorithmically from the Jamo_Short_Name property values for each of the Hangul jamo characters in the full canonical decomposition of that syllable. That derivation is specified here.

Full Canonical Decomposition. First construct the full canonical decomposition *d* for the precomposed Hangul syllable *s*, as specified by the Hangul Syllable Decomposition Algorithm:

```
s → d = <LPart, VPart, (TPart)>
```

Jamo Short Name Mapping. For each part of the full canonical decomposition *d*, look up the Jamo_Short_Name property value, as specified in Jamo.txt in the Unicode Character Database. If there is no TPart in the full canonical decomposition, then the third value is set to be a null string:

```
JSNL = Jamo_Short_Name(LPart)
JSNV = Jamo_Short_Name(VPart)
JSNT = Jamo_Short_Name(TPart)  if TPart exists, else ""
```

Name Concatenation. The Unicode character name for *s* is then constructed by starting with the constant string "HANGUL SYLLABLE" and then concatenating each of the three Jamo short name values, in order:

```
Name = "HANGUL SYLLABLE " + JSN_L + JSN_V + JSN_T
```

Example. For the precomposed Hangul syllable U+D4DB, construct the full canonical decomposition:

```
U+D4DB → <U+1111, U+1171, U+11B6>
```

Look up the Jamo_Short_Name values for each of the Hangul jamo in the canonical decomposition:

```
JSN_L = Jamo_Short_Name(U+1111) = "P"
JSN_V = Jamo_Short_Name(U+1171) = "WI"
JSN_T = Jamo_Short_Name(U+11B6) = "LH"
```

Concatenate the pieces:

```
Name = "HANGUL SYLLABLE " + "P" + "WI" + "LH"
     = "HANGUL SYLLABLE PWILH"
```

Sample Code for Hangul Algorithms

This section provides sample Java code illustrating the three Hangul-related algorithms.

Common Constants. This code snippet defines the common constants used in the methods that follow.

```
static final int
    SBase = 0xAC00,
    LBase = 0x1100, VBase = 0x1161, TBase = 0x11A7,
    LCount = 19, VCount = 21, TCount = 28,
    NCount = VCount * TCount,   // 588
    SCount = LCount * NCount;   // 11172
```

Hangul Decomposition. The Hangul Decomposition Algorithm as specified above directly decomposes precomposed Hangul syllable characters into a sequence of either two or three Hangul jamo characters. The sample method here does precisely that:

```
public static String decomposeHangul(char s) {
    int SIndex = s - SBase;
    if (SIndex < 0 || SIndex >= SCount) {
      return String.valueOf(s);
    }
    StringBuffer result = new StringBuffer();
    int L = LBase + SIndex / NCount;
    int V = VBase + (SIndex % NCount) / TCount;
    int T = TBase + SIndex % TCount;
    result.append((char)L);
    result.append((char)V);
    if (T != TBase) result.append((char)T);
    return result.toString();
  }
```

The Hangul Decomposition Algorithm could also be expressed equivalently as a recursion of binary decompositions, as is the case for other non-Hangul characters. All LVT syllables would decompose into an LV syllable plus a T jamo. The LV syllables themselves would in turn decompose into an L jamo plus a V jamo. This approach can be used to produce somewhat more compact code than what is illustrated in this sample method.

Hangul Composition. An important feature of Hangul composition is that whenever the source string is not in Normalization Form D or Normalization Form KD, one must not detect only character sequences of the form <L, V> and <L, V, T>. It is also necessary to catch the sequences of the form <LV, T>. To guarantee uniqueness, such sequences must also be composed. This extra processing is illustrated in step 2 of the sample method defined here.

```
public static String composeHangul(String source) {
    int len = source.length();
    if (len == 0) return "";
    StringBuffer result = new StringBuffer();
    char last = source.charAt(0);      // copy first char
    result.append(last);

    for (int i = 1; i < len; ++i) {
      char ch = source.charAt(i);

      // 1. check to see if two current characters are L and V
      int LIndex = last - LBase;
      if (0 <= LIndex && LIndex < LCount) {
        int VIndex = ch - VBase;
        if (0 <= VIndex && VIndex < VCount) {

          // make syllable of form LV

          last = (char)(SBase + (LIndex * VCount + VIndex)
            * TCount);

          result.setCharAt(result.length()-1, last); // reset last
          continue; // discard ch
        }
      }

      // 2. check to see if two current characters are LV and T
      int SIndex = last - SBase;
      if (0 <= SIndex && SIndex < SCount
            && (SIndex % TCount) == 0) {
        int TIndex = ch - TBase;
        if (0 < TIndex && TIndex < TCount) {

          // make syllable of form LVT

          last += TIndex;
          result.setCharAt(result.length()-1, last); // reset last
          continue; // discard ch
        }
      }
      // if neither case was true, just add the character
      last = ch;
```

```
      result.append(ch);
    }
    return result.toString();
  }
```

Hangul Character Name Generation. Hangul decomposition is also used when generating the names for precomposed Hangul syllables. This is apparent in the following sample method for constructing a Hangul syllable name. The content of the three tables used in this method can be derived from the data file Jamo.txt in the Unicode Character Database.

```
public static String getHangulName(char s) {
    int SIndex = s - SBase;
    if (0 > SIndex || SIndex >= SCount) {
      throw new IllegalArgumentException("Not a Hangul Syllable: "
        + s);
    }
    int LIndex = SIndex / NCount;
    int VIndex = (SIndex % NCount) / TCount;
    int TIndex = SIndex % TCount;
    return "HANGUL SYLLABLE " + JAMO_L_TABLE[LIndex]
      + JAMO_V_TABLE[VIndex] + JAMO_T_TABLE[TIndex];
  }

  static private String[] JAMO_L_TABLE = {
    "G", "GG", "N", "D", "DD", "R", "M", "B", "BB",
    "S", "SS", "", "J", "JJ", "C", "K", "T", "P", "H"
  };

  static private String[] JAMO_V_TABLE = {
    "A", "AE", "YA", "YAE", "EO", "E", "YEO", "YE", "O",
    "WA", "WAE", "OE", "YO", "U", "WEO", "WE", "WI",
    "YU", "EU", "YI", "I"
  };

  static private String[] JAMO_T_TABLE = {
    "", "G", "GG", "GS", "N", "NJ", "NH", "D", "L", "LG", "LM",
    "LB", "LS", "LT", "LP", "LH", "M", "B", "BS",
    "S", "SS", "NG", "J", "C", "K", "T", "P", "H"
  };
```

Additional Transformations for Hangul Jamo. Additional transformations can be performed on sequences of Hangul jamo for various purposes. For example, to regularize sequences of Hangul jamo into standard Korean syllable blocks, the *choseong* or *jungseong* fillers can be inserted, as described in Unicode Standard Annex #29, "Unicode Text Segmentation."

For keyboard input, additional compositions may be performed. For example, a sequence of trailing consonants $k_f + s_f$ may be combined into a single, complex jamo ks_f. In addition, some Hangul input methods do not require a distinction on input between initial and final consonants, and may instead change between them on the basis of context. For example, in

the keyboard sequence $m_i + e_m + n_i + s_i + a_m$, the consonant n_i would be reinterpreted as n_f, because there is no possible syllable *nsa*. This results in the two syllables *men* and *sa*.

3.13 Default Case Algorithms

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

All of these specifications are *logical* specifications. Particular implementations can optimize the processes as long as they provide the same results.

Tailoring. The default casing operations are intended for use in the *absence* of tailoring for particular languages and environments. Where a particular environment requires tailoring of casing operations to produce correct results, use of such tailoring does not violate conformance to the standard.

Data that assist the implementation of certain tailorings are published in SpecialCasing.txt in the Unicode Character Database. Most notably, these include:

- Casing rules for the Turkish *dotted capital I* and *dotless small i*.
- Casing rules for the retention of dots over *i* for Lithuanian letters with additional accents.

Examples of case tailorings which are not covered by data in SpecialCasing.txt include:

- Titlecasing of IJ at the start of words in Dutch
- Removal of accents when uppercasing letters in Greek
- Titlecasing of second or subsequent letters in words in orthographies that include caseless letters such as apostrophes
- Uppercasing of U+00DF "ß" LATIN SMALL LETTER SHARP S to U+1E9E LATIN CAPITAL LETTER SHARP S

The preferred mechanism for defining tailored casing operations is the Unicode Common Locale Data Repository (CLDR), where tailorings such as these can be specified on a per-language basis, as needed.

Tailorings of case operations may or may not be desired, depending on the nature of the implementation in question. For more about complications in case mapping, see the discussion in *Section 5.18, Case Mappings*.

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 of the latter mappings that would conflict. Any character that does not have a mapping in these files is considered to map to itself. The full case mappings of a character C are referred to as Lowercase_Mapping(C), Titlecase_Mapping(C), and Uppercase_Mapping(C). The full case folding of a character C is referred to as Case_Folding(C).

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

D135 A character C is defined to be *cased* if and only if C has the Lowercase or Uppercase property or has a General_Category value of Titlecase_Letter.

- The Uppercase and Lowercase property values are specified in the data file DerivedCoreProperties.txt in the Unicode Character Database. The derived property Cased is also listed in DerivedCoreProperties.txt.

D136 A character C is defined to be *case-ignorable* if C has the value MidLetter (ML), MidNumLet (MB), or Single_Quote (SQ) for the Word_Break property or its General_Category is one of Nonspacing_Mark (Mn), Enclosing_Mark (Me), Format (Cf), Modifier_Letter (Lm), or Modifier_Symbol (Sk).

- The Word_Break property is defined in the data file WordBreakProperty.txt in the Unicode Character Database.
- The derived property Case_Ignorable is listed in the data file DerivedCoreProperties.txt in the Unicode Character Database.
- The Case_Ignorable property is defined for use in the context specifications of *Table 3-14*. It is a narrow-use property, and is not intended for use in other contexts. The more broadly applicable string casing function, isCased(X), is defined in D143.

D137 *Case-ignorable sequence*: A sequence of zero or more case-ignorable characters.

D138 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 then zero or more case-ignorable characters, and C is not followed by a sequence consisting of zero or more case-ignorable characters 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{ccc=230} \p{ccc=0}])*

Table 3-14. Context Specification for Casing (Continued)

Context	Description	Regular Expressions	
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{ccc=230}\p{ccc=0}]* [\p{ccc=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{ccc=230} \p{ccc=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{ccc=230} \p{ccc=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.

The regular-expression operator * in *Table 3-14* is "possessive," consuming as many characters as possible, with no backup. This is significant in the case of Final_Sigma, because the sets of case-ignorable and cased characters are not disjoint: for example, they both contain U+0345 YPOGEGRAMMENI. Thus, the Before condition is not satisfied if C is preceded by only U+0345, but would be satisfied by the sequence <capital-alpha, ypogegrammeni>. Similarly, the After condition is satisfied if C is only followed by *ypogegrammeni*, but would not satisfied by the sequence <ypogegrammeni, capital-alpha>.

Default Case Conversion

The following rules specify the default case conversion operations for Unicode strings. These rules use the full case conversion operations, Uppercase_Mapping(C), Lowercase_Mapping(C), and Titlecase_Mapping(C), as well as the context-dependent mappings based on the casing context, as specified in *Table 3-14*.

For a string X:

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

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

R3 ***toTitlecase(X): Find the word boundaries in X according to Unicode Standard Annex #29, "Unicode Text Segmentation." For each word boundary, find the first cased character F following the word boundary. If F exists, map F to Titlecase_Mapping(F); then map all characters C between F and the following word boundary to Lowercase_Mapping(C).***

The default case conversion operations may be tailored for specific requirements. A common variant, for example, is to make use of simple case conversion, rather than full case conversion. Language- or locale-specific tailorings of these rules may also be used.

Default Case Folding

Case folding is related to case conversion. However, the main purpose of case folding is to contribute to caseless matching of strings, whereas the main purpose of case conversion is to put strings into a particular cased form.

Default Case Folding does not preserve normalization forms. A string in a particular Unicode normalization form may not be in that normalization form after it has been case-folded.

Default Case Folding is based on the full case conversion operations without the context-dependent mappings sensitive to the casing context. There are also some adaptations specifically to support caseless matching. Lowercase_Mapping(C) is used for most characters, but there are instances in which the folding must be based on Uppercase_Mapping(C), instead. In particular, the addition of lowercase Cherokee letters as of Version 8.0 of the Unicode Standard, together with the stability guarantees for case folding, require that Cherokee letters be case folded to their *uppercase* counterparts. *As a result, a case folded string is not necessarily lowercase.*

Any two strings which are considered to be case variants of each other under any of the full case conversions, toUppercase(X), toLowercase(X), or toTitlecase(X) will fold to the same string by the toCasefold(X) operation:

R4 toCasefold(X): Map each character C in X to Case_Folding(C).

- Case_Folding(C) uses the mappings with the status field value "C" or "F" in the data file CaseFolding.txt in the Unicode Character Database.

A modified form of Default Case Folding is designed for best behavior when doing caseless matching of strings interpreted as identifiers. This folding is based on Case_Folding(C), but also removes any characters which have the Unicode property value Default_Ignorable_Code_Point=True. It also maps characters to their NFKC equivalent sequences. Once the mapping for a string is complete, the resulting string is then normalized to NFC. That last normalization step simplifies the statement of the use of this folding for caseless matching.

R5 toNFKC_Casefold(X): Map each character C in X to NFKC_Casefold(C) and then normalize the resulting string to NFC.

- The mapping NFKC_Casefold (short alias NFKC_CF) is specified in the data file DerivedNormalizationProps.txt in the Unicode Character Database.
- The derived binary property Changes_When_NFKC_Casefolded is also listed in the data file DerivedNormalizationProps.txt in the Unicode Character Database.

For more information on the use of NFKC_Casefold and caseless matching for identifiers, see Unicode Standard Annex #31, "Unicode Identifier and Pattern Syntax."

Default Case Detection

The casing status of a string can be determined by using the casing operations defined earlier. The following definitions provide a specification. They assume that X and Y are strings. In the following, functional names beginning with "is" are binary functions which take the string X and return true when the string as a whole matches the given casing status. For example, isLowerCase(X) would be true if the string X as a whole is lowercase. In contrast, the Unicode character properties such as Lowercase are properties of individual characters.

For each definition, there is also a related Unicode character property which has a name beginning with "Changes_When_". That property indicates whether each character is affected by a particular casing operation; it can be used to optimize implementations of Default Case Detection for strings.

When case conversion is applied to a string that is decomposed (or more precisely, normalized to NFD), applying the case conversion character by character does not affect the normalization status of the string. Therefore, these definitions are specified in terms of Normalization Form NFD. To make the definitions easier to read, they adopt the convention that the string Y equals toNFD(X).

D139 isLowercase(X): isLowercase(X) is true when toLowercase(Y) = Y.

- For example, isLowercase("combining mark") is true, and isLowercase("Combining mark") is false.
- The derived binary property Changes_When_Lowercased is listed in the data file DerivedCoreProperties.txt in the Unicode Character Database.

D140 isUppercase(X): isUppercase(X) is true when toUppercase(Y) = Y.

- For example, isUppercase("COMBINING MARK") is true, and isUppercase("Combining mark") is false.
- The derived binary property Changes_When_Uppercased is listed in the data file DerivedCoreProperties.txt in the Unicode Character Database.

D141 isTitlecase(X): isTitlecase(X) is true when toTitlecase(Y) = Y.

- For example, isTitlecase("Combining Mark") is true, and isTitlecase("Combining mark") is false.
- The derived binary property Changes_When_Titlecased is listed in the data file DerivedCoreProperties.txt in the Unicode Character Database.

D142 isCasefolded(X): isCasefolded(X) is true when toCasefold(Y) = Y.

- For example, isCasefolded("heiss") is true, and isCasefolded("heiß") is false.

- The derived binary property Changes_When_Casefolded is listed in the data file DerivedCoreProperties.txt in the Unicode Character Database.

Uncased characters do not affect the results of casing detection operations such as the string function isLowercase(X). Thus a space or a number added to a string does not affect the results.

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 *simultaneously* 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

Only when a string, such as "123", contains no cased letters will all three conditions,—isLowercase, isUppercase, and isTitlecase—evaluate as true. This combination of conditions can be used to check for the presence of cased letters, using the following definition:

D143 isCased(X): isCased(X) is true when isLowercase(X) is false, or isUppercase(X) is false, or isTitlecase(X) is false.

- Any string X for which isCased(X) is true contains at least one character that has a case mapping other than to itself.
- For example, isCased("123") is false because all the characters in "123" have case mappings to themselves, while isCased("abc") and isCased("A12") are both true.
- The derived binary property Changes_When_Casemapped is listed in the data file DerivedCoreProperties.txt in the Unicode Character Database.

To find out whether a string contains only lowercase letters, implementations need to test for (isLowercase(X) and isCased(X)).

Default Caseless Matching

Default caseless matching is the process of comparing two strings for case-insensitive equality. The definitions of Unicode Default Caseless Matching build on the definitions of Unicode Default Case Folding.

Default Caseless Matching uses full case folding:

D144 A string X is a caseless match for a string Y if and only if:
toCasefold(X) = toCasefold(Y)

When comparing strings for case-insensitive equality, the strings should also be normalized for most correct results. For example, the case folding of U+00C5 Å LATIN CAPITAL LETTER A WITH RING ABOVE is U+00E5 å LATIN SMALL LETTER A WITH RING ABOVE, whereas the case folding of the sequence <U+0041 "A" LATIN CAPITAL LETTER A, U+030A ◌̊ COMBINING RING ABOVE> is the sequence <U+0061 "a" LATIN SMALL LETTER A, U+030A ◌̊ COMBINING RING ABOVE>. Simply doing a binary comparison of the results of case folding both strings will not catch the fact that the resulting case-folded strings are canonical-equivalent sequences. In principle, normalization needs to be done after case folding, because case folding does not preserve the normalized form of strings in all instances. This requirement for normalization is covered in the following definition for canonical caseless matching:

D145 A string X is a canonical caseless match for a string Y if and only if:
NFD(toCasefold(NFD(X))) = NFD(toCasefold(NFD(Y)))

The invocations of canonical decomposition (NFD normalization) before case folding in D145 are to catch very infrequent edge cases. Normalization is not required before case folding, except for the character U+0345 ◌ͅ COMBINING GREEK YPOGEGRAMMENI and any characters that have it as part of their canonical decomposition, such as U+1FC3 ῃ GREEK SMALL LETTER ETA WITH YPOGEGRAMMENI. In practice, optimized versions of canonical caseless matching can catch these special cases, thereby avoiding an extra normalization step for each comparison.

In some instances, implementers may wish to ignore compatibility differences between characters when comparing strings for case-insensitive equality. The correct way to do this makes use of the following definition for compatibility caseless matching:

D146 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))))))

Compatibility caseless matching requires an extra cycle of case folding and normalization for each string compared, because the NFKD normalization of a compatibility character such as U+3392 SQUARE MHZ may result in a sequence of alphabetic characters which must again be case folded (and normalized) to be compared correctly.

Caseless matching for identifiers can be simplified and optimized by using the NFKC_-Casefold mapping. That mapping incorporates internally the derived results of iterated case folding and NFKD normalization. It also maps away characters with the property value Default_Ignorable_Code_Point=True, which should not make a difference when comparing identifiers.

The following defines identifier caseless matching:

D147 A string X is an identifier caseless match for a string Y if and only if:
toNFKC_Casefold(NFD(X)) = toNFKC_Casefold(NFD(Y))