Chapter 3

DRAFT

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 referred to by the conformance clause. These algorithms specify the required results rather than the specific implementation; all implementations that produce results identical to the results of the formal algorithms are conformant.

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

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

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 is conceived of as creating a new repertoire, which then will be given its own catalog number, constituting a new object.

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

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

Implementations should be prepared to be forward-compatible with respect to Unicode versions. That is, they should accept text that may be expressed in future versions of this standard, recognizing that new characters may be assigned in those versions. Thus they should handle incoming unassigned code points as they do unsupported characters. (See *Section 5.3, Unknown and Missing Characters*.)

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

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. If future versions of these implementations or specifications upgrade to a future version of the Unicode Standard, then some changes may be necessary.

Detailed policies on character encoding stability are found on the Unicode Web site. See the subsection on "Stability Policies" in *Section B.4, Other Unicode References.*

Version Numbering

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

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

Additional information on the current and past versions of the Unicode Standard can be found on the Unicode Web site. See the subsection "Versions" in *Section B.4, Other Unicode References.*

References to the Unicode Standard

Properties and property values have defined names and abreviations, 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 defined in Unicode 3.2.0

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

> Unicode 3.2.0 (March, 2002)
>
> The Unicode Consortium. The Unicode Standard, Version 3.2.0, defined by:
>
> *The Unicode Standard, Version 3.0* (Reading, MA, Addison-Wesley, 2000. ISBN 0-201-61633-5), as amended by the Unicode Standard Annex #27: Unicode 3.1 (http://www.unicode.org/reports/tr27/) and the Unicode

Standard Annex #28: Unicode 3.2 (http://www.unicode.org/reports/tr28/)

The reference for this version of the Unicode Standard is:

Unicode 4.0.0 (August, 2003)

The Unicode Consortium. The Unicode Standard, Version 4.0.0, defined by:

The Unicode Standard, Version 4.0 (Reading, MA, Addison-Wesley, 2003. ISBN 0-201-xxxxx-x)

[needs its ISBN and month of 2003 just prior to press 9-8-02]

3.2 Conformance Requirements

This section specifies the formal conformance requirements for processes implementing Version 4.0 of the Unicode Standard. Note that these clauses have been revised from the previous versions of the Unicode Standard. These revisions do not change the fundamental substance of the conformance requirements previously set forth, but rather are reformulated to present a more consistent character encoding model, including the encoding forms. They have also been extended to cover additional aspects of properties, references, and algorithms.

Byte Ordering

C1 [Superseded by C11]

C2 [Superseded by C11]

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

C3 [Superseded by C12b]

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

Unassigned Code Points

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

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

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

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

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

- This clause does not preclude the assignment of certain generic semantics to other 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.
- Code points whose use has not yet been designated may be assigned to abstract characters in future versions of the standard. Because of this fact, due care in the handling of generic semantics for such code points is likely to provide better robustness for implementations that may encounter data based on future versions of the standard.

Interpretation

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

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

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

- 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 real life, any system may occasionally receive an unfamiliar character code that it is unable to interpret.

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

- Ideally, an implementation would always interpret two canonical-equivalent character sequences identically. There are practical circumstances under which implementations may reasonably distinguish them.
- Even processes that normally do not distinguish between canonical-equivalent character sequences can have reasonable exception behavior. Some examples of this behavior include graceful fallback processing by processes unable to support correct positioning of nonspacing marks; "Show Hidden Text" modes that reveal memory representation structure; and the choice of ignoring collating behavior of combining sequences that are not part of the repertoire of a specified language (see *Section 5.13, Strategies for Handling Nonspacing Marks*).

Modification

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

- 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 when transforming between different machine architectures does not modify the interpretation of the text.
- Changing a valid coded character representation 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 which does not have a specific internal use is unexpectedly encountered in processing, an implementation may signal an error or delete or ignore the noncharacter. If these options are not taken, the noncharacter should be treated as an unassigned code point. For example, an API that returned a character property value for a noncharacter would return the same value as the default value for an unassigned code point.

Encoding Forms

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

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

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

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

- For example, in UTF-8 every code unit of the form $110xxxx_2$ *must* be followed by a code unit of the form $10xxxxxx_2$. A sequence such as $110xxxxx_2$ $0xxxxxxx_2$ is ill-formed and must never be generated. When faced with this ill-formed code unit sequence while transforming or interpreting, a conformant process must treat the first code unit $110xxxxx_2$ as an illegal termination error—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. In the latter two cases, it will continue processing at the second code unit $0xxxxxxx_2$.

[Mark to draft new text here: additional examples for Java and C# -- exact location? 8-19-02 ed mtg]

- 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, 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 must not be used in circumstances where it would cause security problems.

Encoding Schemes

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

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

Bidirectional Text

C13 A process that displays text containing supported right-to-left characters or embedding codes shall display all visible representations of characters (excluding format characters) in the same order as if the bidirectional algorithm had been applied to the text, in the absence of higher-level protocols.

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

Normalization Forms

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

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

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

References

C17 Normative references to this standard shall follow the format in Section 3.1, Versions of the Unicode Standard.

C18 Normative references to Unicode properties shall use property aliases, following the format in Section 3.1, Versions of the Unicode Standard.

C19 Normative references to Unicode property values shall use property value aliases, following the format in Section 3.1, Versions of the Unicode Standard.

C20 Higher-level protocols shall not make normative references to provisional properties.

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

Algorithms

C21 *If a process purports to implement a Unicode algorithm, it shall conform to the specification of that algorithm in the standard, unless tailored by a higher-level protocol.*

- The specification of an algorithm may prohibit or limit tailoring by a higher-level protocol.
- Normalization and Canonical Ordering are not tailorable. The bidirectional algorithm allows some tailoring by higher-level protocols.

Unicode Standard Annexes

The following standard annexes are approved and considered part of Version 4.0 of the Unicode Standard. These reports may contain either normative or informative material, or both. Any reference to Version 4.0 of the standard automatically includes these standard annexes.

- UAX #9: The Bidirectional Algorithm, Version 4.0.0
- UAX #11: East Asian Width, Version 4.0.0
- UAX #14: Line Breaking Properties, Version 4.0.0
- UAX #15: Unicode Normalization Forms, Version 4.0.0
- UAX #29: Text Boundaries, Version 4.0.0

3.3 Semantics

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

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

1. Character combination
2. Canonical decomposition
3. Compatibility decomposition
4. Canonical ordering behavior
5. Bidirectional behavior, as specified according to the Unicode bidirectional algorithm (see *Unicode Standard Annex #9, "The Bidirectional Algorithm."*
6. Conjoining jamo behavior, as specified according to *Section 3.12, Conjoining Jamo Behavior*
7. Variation selection, as specified according to *Section 15.6, Variation Selectors*
8. Normalization, as specified in *Unicode Standard Annex #15, "Unicode Normalization Forms."*

D2 [Incorporated into other definitions.]

D2a *Character identity:* The identity of a character is established by its character name and representative glyph in *Chapter 16, Code Charts.*

- A character may have a broader range of use than the most literal interpretation of its name might indicate; the coded representation, name, and representative glyph

need to be taken in context when establishing the identity of a character. For example, U+002E FULL STOP can represent a sentence period, an abbreviation period, a decimal number separator in English, a thousands number separator in German, and so on. The character name itself is unique, but may be misleading. See "Character Names" in *Section 16.1, Character Names List*.

- Consistency with the representative glyph does not require that the images be identical or even graphically similar; rather, it means that both images are generally recognized to be representations of the same character. Representing the character U+0061 LATIN SMALL LETTER A by the glyph "X" would violate its character identity.

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

- Some normative behavior is default behavior; this behavior can be overridden by higher-level protocols. However, in the absence of such protocols, the behavior must be observed so as to follow the character semantics.
- The character combination properties and the canonical ordering behavior cannot be overridden by higher-level protocols.

3.4 Characters and Encoding

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

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

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

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

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

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

- A code point is also known as a *code position*.

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

- An encoded character is also referred to as a *coded character.*

- While an encoded character is formally defined in terms of the mapping between an abstract character and a code point, informally it can be thought of as an abstract character taken together with its assigned code point.
- 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 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>.

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

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

D7 [Incorporated into other definitions.]

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.

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

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

D7b *Noncharacter:* a code point that is permanently reserved for internal use, and that should never be interchanged. These 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 15.8, Noncharacters.*
- These code points are permanently reserved as noncharacters.

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

- Note that surrogate code points and noncharacters are considered assigned code points, but not assigned characters.

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

D8 *Higher-level protocol:* any agreement on the interpretation of Unicode characters that extends beyond the scope of this standard. Such an agreement need not be formally announced in data; it may be implicit in the context.

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

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

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.

Normative and Informative Properties

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

D9 *Normative property:* A Unicode character property whose values are required for conformance to 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. Thus, for example, the Directionality property (bidirectional character type) is required for conformance whenever rendering text which requires bidirectional layout, such as Arabic or Hebrew.

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.

Some of the normative properties are *immutable*, which means that they disallow any kind of overriding by higher-level protocols. Thus, for example, the decomposition of Unicode characters is normative *and* immutable; 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 provide a common basis for behavior, but they may require tailoring for particular local cultural conventions or particular implementations.

The normative character properties of the Unicode Standard are listed in *Table 3-1*.

Table 3-1. Normative Character Properties

Property	Description
Case	Chapter 4 and Chapter 16
Case Mapping	*Section 5.19*
Combining Classes	Chapter 3 and UAX #15
Decomposition (Canonical and Compatibility)	Chapter 3, Chapter 16, and UAX #15
Directionality	UAX #9
Jamo Short Name	Chapter 3
Numeric Value	Chapter 4
Special Character Properties	Chapter 15 and *Section 3.11*
Mirrored	Chapter 3 and UAX #9
Unicode Character Names	Chapter 16

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

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

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 3-2 provides a partial list of the more important informative character properties. For a complete listing, see the Unicode Character Database.

Table 3-2. Informative Character Properties

Property	Description
Dashes	Chapter 6 and *Table 6-3*
East Asian Width	*Section 11.3* and UAX #11
Letters (Alphabetic and Ideographic)	*Section 4.9*
Line Breaking	*Section 15.1*, *Section 15.2*, and UAX #14
Mathematical Property	*Section 14.4*
Spaces	*Table 6-2*
Unicode 1.0 Names	*Section 4.8*

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

Some of the information provided about characters in the Unicode Character Database constitutes provisional data. This may capture partial or preliminary information. It may contain errors or omissions, or otherwise not be ready for systematic use; however, it is

included in the data files for distribution partly to encourage review and improvement of the information. For example, a number of the tags in the Unihan.txt file provide provisional property values of various sorts about Han characters.

The data files of the Unicode Character Database may also contain various annotations and comments about characters, and those annotations and comments should also 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.

Simple and Derived Properties

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

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

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

Derived properties that are computed solely from normative simple properties are themselves considered normative; other derived properties are considered informative.

Property Aliases

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

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

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

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

- The identifiers used for property value aliases also contain only ASCII alphanumeric characters or the underscore character.
- Property value aliases are defined in the file PropertyValueAliases.txt in the Unicode Character Database.
- Property value aliases are only unique identifiers in the context of the particular property they are associated with. 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”).

Default Property Values

Implementations need specific property values for all code points, including those code points that are unassigned. To meet this need, the Unicode Standard assigns default properties to ranges of unassigned code points.

D11 *Default property value:* For a given Unicode property, the value of that property which is assigned, by default, to unassigned code points or to code points not explicitly specified to have other values of that property.

In some instances the default property value is an implied null value. For example, there is no Unicode Character Name for unassigned code points, which is equivalent to saying that the Unicode Character Name of an unassigned code point is a null string.

In other instances, the documentation regarding a particular property is very explicit, and provides a specific enumerated value for the default property value. For example, “XX” is the explicit default property value for the Line Break property.

Private Use

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

- Private-use code points are considered to be assigned characters, but the abstract characters associated with them have no interpretation specified by this standard. They can be given any interpretation by conformant processes.

3.6 Combination

D13 *Base character:* a character that does not graphically combine with preceding characters, and that is neither a control nor a format character.

- Most Unicode characters are base characters. This sense of graphic combination does not preclude the presentation of base characters from adopting different contextual forms or participating in ligatures.

D14 *Combining character:* a character that graphically combines with a preceding base character. The combining character is said to *apply* to that base character.

- These characters are not used in isolation (unless they are being described). They include such characters as accents, diacritics, Hebrew points, Arabic vowel signs, and Indic matras.
- Even though a combining character is intended to be presented in graphical combination with a base character, circumstances may arise where either (1) no base character precedes the combining character or (2) a process is unable to perform graphical combination. In both cases, a process may present a combining character

without graphical combination; that is, it may present it as if it were a base character.

- The representative images of combining characters are depicted with a dotted circle in the code charts; when presented in graphical combination with a preceding base character, that base character is intended to appear in the position occupied by the dotted circle.
- Combining characters generally take on the properties of their base character, while retaining their combining property.
- Control and format characters, such as *tab* or *right-left mark*, are not base characters. Combining characters do not apply to them.

D15 *Nonspacing mark:* a combining character whose positioning in presentation is dependent on its base character. It generally does not consume space along the visual baseline in and of itself.

- Such characters may be large enough to affect the placement of their base character relative to preceding and succeeding base characters. For example, a circumflex applied to an "i" may affect spacing ("î"), as might the character U+20DD COMBINING ENCLOSING CIRCLE.

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

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

D17 *Combining character sequence:* a character sequence consisting of either a base character followed by a sequence of one or more combining characters, or a sequence of one or more combining characters.

- A combining character sequence is also referred to as a *composite character sequence.*

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

- Defective combining character sequences occur when a sequence of combining characters appears at the start of a string or follows a control or format character.

3.7 Decomposition

D18 *Decomposable character:* a character that is equivalent to a sequence of one or more other characters, according to the decomposition mappings found in the names list of *Section 16.1, Character Names List*. It may also be known as a *precomposed* character or *composite* character.

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

Compatibility Decomposition

D20 *Compatibility decomposition:* the decomposition of a character that results from recursively applying *both* the compatibility mappings *and* the canonical mappings found in the names list of *Section 16.1, Character Names List*, and those described in *Section 3.12, Conjoining Jamo Behavior*, until no characters can be further decom-

posed, and then reordering nonspacing marks according to *Section 3.11, Canonical Ordering Behavior*.

- A compatibility decomposition may remove formatting information.

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

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

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

Canonical Decomposition

D23 *Canonical decomposition:* the decomposition of a character that results from recursively applying the canonical mappings found in the names list of *Section 16.1, Character Names List*, and those described in *Section 3.12, Conjoining Jamo Behavior*, until no characters can be further decomposed, and then reordering nonspacing marks according to *Section 3.11, Canonical Ordering Behavior*.

- A canonical decomposition does not remove formatting information.

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

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

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

- For example, the sequences <*o, combining-diaeresis*> and <*ö*> are canonical equivalents. Canonical equivalence is a Unicode property. It should not be confused with language-specific collation or matching, which may add additional equivalencies. For example, in Swedish, *ö* is treated as a completely different letter from *o*, collated after *z*. In German, *ö* is weakly equivalent to *oe* and collated with *oe*. In English, *ö* is just an *o* with a diacritic that indicates that it is pronounced separately from the previous letter (as in *coöperate*) and is collated with *o*.
- By definition all canonical equivalent sequences are also compatibility equivalent sequences.

Note: For definitions of *canonical composition* and *compatibility composition*, see Unicode Standard Annex #15, "Unicode Normalization Forms."

3.8 Surrogates

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

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

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

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

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

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

- Surrogate pairs are used only in UTF-16. (See *Section 3.9, Unicode Encoding Forms*.)
- Isolated surrogate code units have no interpretation on their own. Certain other isolated code units in other encoding forms also have no interpretation on their own. For example, the isolated byte 80_{16} has no interpretation in UTF-8; it can *only* be used as part of a multibyte sequence.
- 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 15.5, 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 a defined range of Unicode code points to code unit sequences. The size of the code unit is specified for each encoding form. This section presents the formal definition of each of these encoding forms.

D28 *Unicode scalar value:* any 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.

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

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

D28b *Code unit sequence:* an ordered sequence of code units.

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

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

- For historical reasons, the Unicode encoding forms are also referred to as *Unicode* (or *UCS*) *transformation formats* (UTF). That term is, however, ambiguous between its usage for encoding forms and encoding schemes.
- The mapping of the set of Unicode scalar values to the set of code unit sequences for a Unicode encoding form is *one-to-one*. This property guarantees that a reverse mapping can always be derived. Given the mapping of any Unicode scalar value to a particular code unit sequence for a given encoding form, one can derive the original Unicode scalar value unambiguously from that code unit sequence.

- The mapping of the set of Unicode scalar values to the set of code unit sequences for a Unicode encoding form is not *onto.* In other words for any given encoding form, there exist code unit sequences that have no associated Unicode scalar value.
- To ensure that the mapping for a Unicode encoding form is one-to-one, *all* Unicode scalar values, including those corresponding to noncharacter code points and unassigned code points, must be mapped to unique code unit sequences. Note that this requirement does not extend to high-surrogate and low-surrogate code points, which are excluded by definition from the set of Unicode scalar values.

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

- In the rawest form, Unicode strings may be implemented simply as arrays of the appropriate integral data type, consisting of a sequence of code units lined up one immediately after the other.
- Code units of different Unicode encoding forms must not be mixed in a single Unicode string.

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

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

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

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

- Any code unit sequence that would correspond to a code point outside the defined range of Unicode scalar values would, for example, be ill-formed.
- Also, 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, hence resulting in an ill-formed code unit sequence.

[Mark to add 3rd bullet w/ example for Unicode strings. 8-19-02]

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

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

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

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

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

D30e 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 simply 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 simply 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 simply 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 which takes the two Unicode 16-bit strings, <004D D800> and <DF02 004D>, each of which contains an ill-formed UTF-16 code unit sequence, and concatenates them to form another Unicode string <004D D800 DF02 004D>, which contains a well-formed UTF-16 code unit sequence. The first two Unicode strings are not *in* UTF-16, but the resultant Unicode string is.
- As another example, the code unit sequence <C0 80 61 F3> is a Unicode 8-bit string, but does not consist of a well-formed UTF-8 code unit sequence. That code unit sequence could not result from the specification of the UTF-8 encoding form, and is thus ill-formed. (The same code unit sequence could, of course, be well-formed in the context of some other character encoding standard using 8-bit code units, such as ISO/IEC 8859-1, or vendor code pages.)

If, on the other hand, a Unicode string *purports* to be *in* a Unicode encoding form, then it must contain only a well-formed code unit sequence. If there is an ill-formed code unit sequence in a source Unicode string, then a conformant process which verifies that the Unicode string is in a Unicode encoding form must reject the ill-formed code unit sequence. (See conformance clause C12.) For more information, see *Section 2.6, Unicode Strings*.

Table 3-3 gives an example which summarizes the three Unicode encoding forms.

Table 3-3. Summary 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	D1 90
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 80 8C 82

UTF-32

D31 *UTF-32 encoding form:* the Unicode encoding form which assigns each Unicode scalar value to a single 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*.

D32 [superseded]

D33 [incorporated in other definitions]

D34 [incorporated in other definitions]

UTF-16

D35 *UTF-16 encoding form:* the Unicode encoding form which assigns each Unicode scalar value in the ranges U+0000..U+D7FF and U+E000..U+FFFF to a single 16-bit code unit with the same numeric value as the Unicode scalar value, and which assigns each Unicode scalar value in the ranges U+10000..U+10FFFF to a surrogate pair, according to *Table 3-4*.

- 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 3-4 specifies the bit distribution for the UTF-16 encoding form. Note that for Unicode scalar values equal to or greater than U+10000, UTF-16 uses surrogate pairs. Calculation of the surrogate pair values involves subtraction of 10000_{16}, to account for the starting offset to the scalar value. ISO/IEC 10646 specifies an equivalent UTF-16 encoding form. For details, see *Section C.3, UCS Transformation Formats*.

Table 3-4. UTF-16 Bit Distribution

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

Where wwww = uuuuu - 1.

UTF-8

D36 *UTF-8 encoding form:* the Unicode encoding form which assigns each Unicode scalar value to a byte sequence of one to four bytes in length, as specified in *Table 3-5*.

- In UTF-8, the code point sequence <004D, 0430, 4E8C, 10302> is represented as <4D D1 90 E4 BA 8C F0 80 8C 82>, where <4D> corresponds to U+004D, <D1 90>

corresponds to U+0430, <E4 BA 8C> corresponds to U+4E8C, and <F0 80 8C 82> corresponds to U+10302.

- Any UTF-8 byte sequence that does not match the patterns listed in *Table 3-6* 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 3-6*.
- Because surrogate code points are not Unicode scalar values, any UTF-8 byte sequence that would otherwise map to code points D800..DFFF is ill-formed.

Table 3-5 specifies the bit distribution for the UTF-8 encoding form, showing the ranges of Unicode scalar values corresponding to one-, two-, three-, or four-byte sequences. For a discussion of the difference in the formulation of UTF-8 in ISO/IEC 10646, see *Section C.3, UCS Transformation Formats.*

Table 3-5. UTF-8 Bit Distribution

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

Table 3-6 lists all of the byte sequences that are well-formed in UTF-8. A range of byte values such as A0..BF indicates that any byte from A0 to BF (inclusive) is well-formed in that position. Any byte value outside of the ranges listed is ill-formed. For example:

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

Table 3-6. Well-formed UTF-8 Byte Sequences

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

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

D36a [incorporated in other definitions]

D36b [superseded]

Encoding Form Conversion

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

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

3.10 Unicode Encoding Schemes

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

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

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

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

- In the UTF-8 encoding scheme, the UTF-8 code unit sequence <4D D1 90 E4 BA 8C F0 80 8C 82> is serialized as <4D D1 90 E4 BA 8C F0 80 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 is not considered non-conformant for the UTF-8 encoding scheme. Identification of the <EF BB BF> byte sequence at the beginning of a data stream can, however, be taken as near-certain indication that the data stream is using the UTF-8 encoding scheme.

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

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

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

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

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

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

Table 3-7 gives an example which summarizes the three Unicode encoding schemes for the UTF-16 encoding form.

Table 3-7. Summary of UTF-16BE, UTF-16LE, and UTF-16

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

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

- In UTF-32BE, the UTF-32 code unit sequence <0000004D 00000430 00004E8C 00010302> is serialized as <00 00 00 4D 00 00 04 30 00 00 4D 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.

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

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

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

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

Table 3-8 gives an example which summarizes the three Unicode encoding schemes for the UTF-32 encoding form.

Table 3-8. Summary of UTF-32BE, UTF-32LE, and UTF-32

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

Table 3-8. Summary of UTF-32BE, UTF-32LE, and UTF-32

00010302	UTF-32BE	00 01 03 02
	UTF-32LE	02 03 01 00
	UTF-32	00 00 FE FF 00 01 03 02 FF FE 00 00 02 03 01 00 00 01 03 02

Note that the terms *UTF-8*, *UTF-16*, and *UTF-32*, when used unqualified, are ambiguous between their sense as a Unicode encoding form or a Unicode encoding scheme. 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, on the other hand, 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 converting between different encoding schemes, extreme care must be taken in handling any initial byte order marks. For example, if one converted a UTF-16 byte serialization with an initial byte order mark to a UTF-8 byte serialization, converting the byte order mark to <EF BB BF> in the UTF-8 form, the <EF BB BF> would now be ambiguous as to its status as a byte order mark (from its source) or as an initial *zero width no-break space*. If the UTF-8 byte serialization were then converted to a UTF-16BE and the initial <EF BB BF> were converted to <FE FF>, the interpretation of the U+FEFF would have been modified by the conversion. This would be nonconformant by conformance clause C10, because the change between byte serializations would have resulted in modification of the interpretation of the text. This is one of the reasons why the use of initial <EF BB BF> as a signature on UTF-8 byte sequences is not recommended by the Unicode Standard.

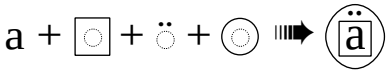
3.11 Canonical Ordering Behavior

The purpose of this section is to provide unambiguous interpretation of a combining character sequence. Normalization is an important application of canonical ordering behavior. See Unicode Standard Annex #15, “Unicode Normalization Forms.”

In the Unicode Standard, the order of characters in a combining character sequence is interpreted according to the following principles:

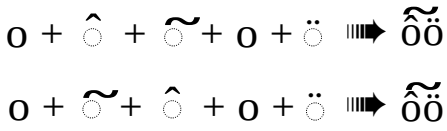
- In the Unicode Standard, all combining characters are encoded following the base characters to which they apply. Thus the Unicode sequence U+0061 “a” LATIN SMALL LETTER A + U+0308 “◌̈” COMBINING DIAERESIS + U+0075 “u” LATIN SMALL LETTER U is unambiguously interpreted (and displayed) as “äu”, not “aü”.
- Enclosing nonspacing marks surround all previous characters up to and including the base character (see *Figure 3-1*). They thus successively surround previous enclosing nonspacing marks.

Figure 3-1. Enclosing Marks



- Double diacritics always bind more loosely than other nonspacing marks. When rendering, the double diacritic will float above other diacritics, excluding enclosing diacritics (see *Figure 3-2*).

Figure 3-2. Positioning of Double Diacritics



[Need clarification for behavior of dots above double diacritics. Ken to do. JDA 9-29-02]

- Combining marks with the same combining class are generally positioned graphically outward from the base character they modify. Some specific nonspacing marks override the default stacking behavior by being positioned side-by-side rather than stacking or by ligaturing with an adjacent nonspacing mark. When positioned side-by-side, the order of codes is reflected by positioning in the dominant order of the script with which they are used.
- If combining characters have different combining classes—for example, when one nonspacing mark is above a base character form and another is below it—then no distinction of graphic form or semantic will result.

The following subsections formalize these principles in terms of a normative list of combining classes and an algorithmic statement of how to use those combining classes to unambiguously interpret a combining character sequence.

Application of Combining Marks

Formally speaking, combining marks apply to the preceding grapheme cluster. In most cases, this is the same as applying to the preceding base character. However, in two circumstances there is a difference:

- Hangul syllables
- Enclosing combining marks

Hangul Syllables. Where a grapheme cluster contains a Hangul syllable, the combining mark applies to the entire syllable. For example, in the following sequence the *grave* is applied to the entire Hangul syllable, not just the last jamo:

- U+1100 HANGUL CHOSEONG KIYEOK
- U+1161 HANGUL JUNGSEONG A
- U+0300 COMBINING GRAVE ACCENT

Enclosing Combining Marks. These marks enclose an entire preceding grapheme cluster. For example, in the following sequence the entire Hangul syllable (a single grapheme cluster consisting of three characters) is circled, not just part of it:

- U+1100 HANGUL CHOSEONG KIYEOK
- U+1161 HANGUL JUNGSEONG A
- U+20DD COMBINING ENCLOSING CIRCLE

These marks also enclose as a whole any preceding sequence of grapheme clusters linked by a Grapheme_Link or *combining grapheme joiner*. For example, the entire conjunct is circled in the following sequence:

- U+0915 DEVANAGARI LETTER KA
- U+094D DEVANAGARI SIGN VIRAMA
- U+0922 DEVANAGARI LETTER DDHA
- U+20DD COMBINING ENCLOSING CIRCLE

On the other hand, non-enclosing combining marks only apply to the last base character, irrespective of whether there are Grapheme_Link or *combining grapheme joiner* characters. For example, in the following sequence the *nukta* applies to the immediately preceding *ddha*, not to the entire cluster:

- U+0915 DEVANAGARI LETTER KA
- U+094D DEVANAGARI SIGN VIRAMA
- U+0922 DEVANAGARI LETTER DDHA
- U+093C DEVANAGARI SIGN NUKTA

For more information, see the subsection on "Combining Grapheme Joiner" in *Section 15.2, Layout Controls*.

Combining Classes

The Unicode Standard treats sequences of nonspacing marks as equivalent if they do not typographically interact. The canonical ordering algorithm defines a method for determining which sequences interact and gives a canonical ordering of these sequences for use in equivalence comparisons.

D46 *Combining class:* a numeric value given to each combining Unicode character that determines with which other combining characters it typographically interacts.

- See *Section 4.3, Combining Classes—Normative*, for a list of the combining classes for Unicode characters.

Characters have the same class if they interact typographically, and different classes if they do not.

- Enclosing characters and spacing combining characters have the class of their base characters.
- The particular numeric value of the combining class does not have any special significance; the intent of providing the numeric values is *only* to distinguish the combining classes as being different, for use in equivalence comparisons.

Canonical Ordering

The canonical ordering of a decomposed character sequence results from a sorting process that acts on each sequence of combining characters according to their combining class. Characters with combining class zero never sort relative to other characters, so the amount of work in the algorithm depends on the number of non-class-zero characters in a row. An implementation of this algorithm will be extremely fast for typical text.

The algorithm described here represents a logical description of the process. Optimized algorithms can be used in implementations as long as they are equivalent—that is, as long

as they produce the same result. This algorithm is not tailorable; higher-level protocols shall not specify different results.

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

R1 ***For each character x in D, let p(x) be the combining class of x.***

R2 ***Whenever any pair (A, B) of adjacent characters in D is such that p(B) ≠ 0 & p(A) > p(B), exchange those characters.***

R3 ***Repeat step R2 until no exchanges can be made among any of the characters in D.***

Examples of this ordering appear in *Table 3-9*.

Table 3-9. Sample Combining Classes

Combining Class	Abbreviation	Code	Unicode Name
0	a	0061	LATIN SMALL LETTER A
220	underdot	0323	COMBINING DOT BELOW
230	diaeresis	0308	COMBINING DIAERESIS
230	breve	0306	COMBINING BREVE
0	a-underdot	1EA1	LATIN SMALL LETTER A WITH DOT BELOW
0	a-diaeresis	00E4	LATIN SMALL LETTER A WITH DIAERESIS
0	a-breve	0103	LATIN SMALL LETTER A WITH BREVE

a + underdot + diaeresis ≡ *a + underdot + diaeresis*

a + diaeresis + underdot ≡ *a + underdot + diaeresis*

Because *underdot* has a lower combining class than *diaeresis*, the algorithm will return the *a*, then the *underdot*, then the *diaeresis*. However, because *diaeresis* and *breve* have the same combining class (because they interact typographically), they do not rearrange.

a + breve + diaeresis ≢ *a + diaeresis + breve*

a + diaeresis + breve ≢ *a + breve + diaeresis*

Applying the algorithm gives the results shown in *Table 3-10*.

Table 3-10. Canonical Ordering Results

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

Use with Collation

When collation processes do not require correct sorting outside of a given domain, they are not required to invoke the canonical ordering algorithm for excluded characters. For example, a Greek collation process may not need to sort Cyrillic letters properly; in that case, it does not have to maximally decompose and reorder Cyrillic letters and may just choose to

sort them according to Unicode order. For the complete treatment of collation, see Unicode Technical Standard #10, "Unicode Collation Algorithm."

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 syllable blocks as well as modern syllable blocks. This section describes how to

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

For more information, see the "Hangul Syllables" and "Hangul Jamo" subsections in *Section 11.4, Hangul*. Hangul syllables are a special case of grapheme clusters.

The jamo characters can be classified into three sets of characters: *choseong* (leading consonants, or syllable-initial characters), *jungseong* (vowels, or syllable-peak characters), and *jongseong* (trailing consonants, or syllable-final characters). In the following discussion, these jamo are abbreviated as L (leading consonant), V (vowel), and T (trailing consonant); syllable breaks are shown by *middle dots* "·"; non-syllable breaks are shown by "×", combining marks are shown by M, and non-jamo are shown by X.

In the following discussion, a *syllable* refers to a sequence of Korean characters that should be grouped into a single cell for display. This is different from a *precomposed Hangul syllable*, which consists of any of the characters in the range U+AC00..U+D7A3. Note that a syllable may contain a precomposed Hangul syllable *plus* other characters.

Syllable Boundaries

In rendering, a sequence of jamos is displayed as a series of syllable blocks. The following rules specify how to divide up an arbitrary sequence of jamos (including nonstandard sequences) into these syllable blocks. In these rules, a *choseong filler* (L_f) is treated as a *choseong* character, and a *jungseong filler* (V_f) is treated as a *jungseong*.

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

Within any sequence of characters, a syllable break never occurs between the pairs of characters shown in *Table 3-11*. In all other cases, there is a syllable break before and after any jamo or precomposed Hangul syllable. Note that like other characters, any combining mark between two conjoining jamos prevents the jamos from forming a syllable.

Table 3-11. Hangul Syllable No-Break Rules

Do Not Break Between		Examples
L	L, V, or precomposed Hangul syllable	L × L L× V L × LV L × LVT

Table 3-11. Hangul Syllable No-Break Rules (Continued)

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

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

For information on the behavior of Hangul compatibility jamo in syllables, see *Section 11.4, Hangul*.

Standard Korean Syllables

A standard Korean syllable block is composed of a sequence of one or more *L* followed by a sequence of one or more *V* and optionally a sequence of zero or more *T*. A sequence of nonstandard syllable blocks can be transformed into a sequence of standard Korean syllable blocks by inserting *choseong* fillers (L_f) and *jungseong* fillers (V_f).

Using regular expression notation, a standard Korean syllable is thus of the form:

*L+ V+ T**

The transformation of a string of text into standard Korean syllables is performed by determining the syllable breaks as explained in the subsection on "Syllable Boundaries" earlier in this section, then inserting one or two fillers as necessary to transform each syllable into a standard Korean syllable. Thus:

$L\ \hat{}V \rightarrow L\ V_f\ \hat{}V$

$\hat{}L\ V \rightarrow \hat{}L\ L_f\ V$

$\hat{}V\ T \rightarrow \hat{}V\ L_f\ V_f\ T$

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

[Format above may need tweaking JDA 9-29-02]

Examples. In *Table 3-12*, the first row shows syllable breaks in a standard sequence, the second row shows syllable breaks in a nonstandard sequence, and the third row shows how the sequence in the second row could be transformed into standard form by inserting fillers into each syllable.

Table 3-12. Syllable Break Examples

No.	Sequence		Sequence with Syllable Breaks Marked
1	LVTLVLVLV$_f$L$_f$VL$_f$V$_f$T	→	LVT · LV · LV · LV$_f$ · L$_f$V · L$_f$V$_f$T
2	LLTTVVTTVVLLVV	→	LL · TT · VVTT · VV · LL · LLVV

Table 3-12. Syllable Break Examples

3	LLTTVVTTVVLLVV	→	$LLV_f \cdot L_fV_fTT \cdot L_fVVTT \cdot L_fVV \cdot LLV_f \cdot LLVV$

Hangul Syllable Composition

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

First, define the following constants:

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

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

[Get real greater than or equal signs above to replace <=, >=. JDA 9-29-02]

   ```
   TIndex = 0
   j      = i + 1
   ```
5. Replace the characters D[*i*] through D[*j*] by the Hangul syllable S, and set *i* to be *j*+1.
   ```
   S      = (LIndex * VCount + VIndex) * TCount + TIndex + SBase.
   ```

Example. The first three characters are

U+1111 ᄑ HANGUL CHOSEONG PHIEUPH

U+1171 ᅱ HANGUL JUNGSEONG WI

U+11B6 ᆶ HANGUL JONGSEONG RIEUL-HIEUH

Compute the following indices,

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

and replace the three characters by

```
S      = [(17 * 21) + 16] * 28 + 15 + SBase
       = D4DB16
       =
```

Hangul Syllable Decomposition

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

1. Compute the index of the syllable:
   ```
   SIndex = S - SBase
   ```

[Fix less than or equal to sign below to be in real font. JDA 9-29-02]

2. If *SIndex* is in the range (0 <= *SIndex* < *SCount*), then compute the components as follows:
   ```
   L      = LBase + SIndex / NCount
   V      = VBase + (SIndex % NCount) / TCount
   T      = TBase + SIndex % TCount
   ```
   The operators "/" and "%" are as defined in *Section 0.3, Notational Conventions*.

3. If *T = TBase*, then there is no trailing character, so replace *S* by the sequence <*L*, *V*>. Otherwise, there is a trailing character, so replace *S* by the sequence <*L*, *V*, *T*>.

Example

```
L      = LBase + 17
V      = VBase + 16
T      = TBase + 15
D4DB16 → 111116, 117116, 11B616
```

Hangul Syllable Names

The character names for Hangul syllables are derived from the decomposition by starting with the string HANGUL SYLLABLE, and appending the short name of each decomposition component in order. (See *Chapter 16, Code Charts* and the Unicode Character Database.) For example, for U+D4DB, derive the decomposition, as shown in the preceding example. It produces the following three-character sequence:

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

The character name for U+D4DB is then generated as HANGUL SYLLABLE PWILH. This character name is a normative property of the character.