The Unicode® Standard
Version 12.0 – 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.

© 2019 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 12.0.

Includes index.

ISBN 978-1-936213-22-1 (http://www.unicode.org/versions/Unicode12.0.0/)

1. Unicode (Computer character set) I. Unicode Consortium.

QA268.U545 2019

ISBN 978-1-936213-22-1

Published in Mountain View, CA

March 2019

Chapter 4
Character Properties

> ***Disclaimer***
>
> The content of all character property tables has been verified as far as possible by the Unicode Consortium. However, in case of conflict, the most authoritative version of the information for this version of the Unicode Standard is that supplied in the Unicode Character Database on the Unicode website. *The contents of all the tables in this chapter may be superseded or augmented by information in future versions of the Unicode Standard.*

The Unicode Standard associates a rich set of semantics with characters and, in some instances, with code points. The support of character semantics is required for conformance; see *Section 3.2, Conformance Requirements*. Where character semantics can be expressed formally, they are provided as machine-readable lists of character properties in the Unicode Character Database (UCD). This chapter gives an overview of character properties, their status and attributes, followed by an overview of the UCD and more detailed notes on some important character properties. For a further discussion of character properties, see Unicode Technical Report #23, "Unicode Character Property Model."

Status and Attributes. Character properties may be normative, informative, contributory, or provisional. Normative properties are those required for conformance. Many Unicode character properties can be overridden by implementations as needed. *Section 3.2, Conformance Requirements*, specifies when such overrides must be documented. A few properties, such as Noncharacter_Code_Point, may not be overridden. See *Section 3.5, Properties*, for the formal discussion of the status and attributes of properties.

Consistency of Properties. The Unicode Standard is the product of many compromises. It has to strike a balance between uniformity of treatment for similar characters and compatibility with existing practice for characters inherited from legacy encodings. Because of this balancing act, one can expect a certain number of anomalies in character properties. For example, some pairs of characters might have been treated as canonical equivalents but are left unequivalent for compatibility with legacy differences. This situation pertains to U+00B5 µ MICRO SIGN and U+03BC μ GREEK SMALL LETTER MU, as well as to certain Korean jamo.

In addition, some characters might have had properties differing in some ways from those assigned in this standard, but those properties are left as is for compatibility with existing practice. This situation can be seen with the halfwidth voicing marks for Japanese

(U+FF9E HALFWIDTH KATAKANA VOICED SOUND MARK and U+FF9F HALFWIDTH KATAKANA SEMI-VOICED SOUND MARK), which might have been better analyzed as spacing combining marks. Another examples consists of the conjoining Hangul jamo, which might have been better analyzed as an initial base character followed by formally combining medial and final characters. In the interest of efficiency and uniformity in algorithms, implementations may take advantage of such reanalyses of character properties, as long as this does not conflict with the conformance requirements with respect to normative properties. See *Section 3.5, Properties*; *Section 3.2, Conformance Requirements*; and *Section 3.3, Semantics*, for more information.

4.1 Unicode Character Database

The Unicode Character Database (UCD) consists of a set of files that define the Unicode character properties and internal mappings. For each property, the files determine the assignment of property values to each code point. The UCD also supplies recommended property aliases and property value aliases for textual parsing and display in environments such as regular expressions.

The properties include the following:

- Name
- General Category (basic partition into letters, numbers, symbols, punctuation, and so on)
- Other important general characteristics (whitespace, dash, ideographic, alphabetic, noncharacter, deprecated, and so on)
- Display-related properties (bidirectional class, shaping, mirroring, width, and so on)
- Casing (upper, lower, title, folding—both simple and full)
- Numeric values and types
- Script and Block
- Normalization properties (decompositions, decomposition type, canonical combining class, composition exclusions, and so on)
- Age (version of the standard in which the code point was first designated)
- Boundaries (grapheme cluster, word, line, and sentence)

See Unicode Standard Annex #44, "Unicode Character Database," for more details on the character properties and their values, the status of properties, their distribution across data files, and the file formats.

Unihan Database. In addition, a large number of properties specific to CJK ideographs are defined in the Unicode Character Database. These properties include source information, radical and stroke counts, phonetic values, meanings, and mappings to many East Asian standards. The values for all these properties are listed in the file Unihan.zip, also known as the *Unihan Database*. For a complete description and documentation of the properties themselves, see Unicode Standard Annex #38, "Unicode Han Database (Unihan)." (See also "Online Unihan Database" in *Section B.3, Other Unicode Online Resources*.)

Many properties apply to both ideographs and other characters. These are not specified in the Unihan Database.

Stability. While the Unicode Consortium strives to minimize changes to character property data, occasionally character properties must be updated. When this situation occurs, a new version of the Unicode Character Database is created, containing updated data files.

Data file changes are associated with specific, numbered versions of the standard; character properties are never silently corrected between official versions.

Each version of the Unicode Character Database, once published, is absolutely stable and will never change. Implementations or specifications that refer to a specific version of the UCD can rely upon this stability. Detailed policies on character encoding stability as they relate to properties are found on the Unicode website. See the subsection “Policies” in *Section B.3, Other Unicode Online Resources*. See also the discussion of versioning and stability in *Section 3.1, Versions of the Unicode Standard*.

Aliases. Character properties and their values are given formal aliases to make it easier to refer to them consistently in specifications and in implementations, such as regular expressions, which may use them. These aliases are listed exhaustively in the Unicode Character Database, in the data files PropertyAliases.txt and PropertyValueAliases.txt.

Many of the aliases have both a long form and a short form. For example, the General Category has a long alias “General_Category” and a short alias “gc”. The long alias is more comprehensible and is usually used in the text of the standard when referring to a particular character property. The short alias is more appropriate for use in regular expressions and other algorithmic contexts.

In comparing aliases programmatically, loose matching is appropriate. That entails ignoring case differences and any whitespace, underscore, and hyphen characters. For example, “GeneralCategory”, “general_category”, and “GENERAL-CATEGORY” would all be considered equivalent property aliases. See Unicode Standard Annex #44, “Unicode Character Database,” for further discussion of property and property value matching.

For each character property whose values are not purely numeric, the Unicode Character Database provides a list of value aliases. For example, one of the values of the Line_Break property is given the long alias “Open_Punctuation” and the short alias “OP”.

Property aliases and property value aliases can be combined in regular expressions that pick out a particular value of a particular property. For example, “\p{lb=OP}” means the Open_Punctuation value of the Line_Break property, and “\p{gc=Lu}” means the Uppercase_Letter value of the General_Category property.

Property aliases define a namespace. No two character properties have the same alias. For each property, the set of corresponding property value aliases constitutes its own namespace. No constraint prevents property value aliases for *different* properties from having the same property value alias. Thus “B” is the short alias for the Paragraph_Separator value of the Bidi_Class property; “B” is also the short alias for the Below value of the Canonical_Combining_Class property. However, because of the namespace restrictions, any combination of a property alias plus an appropriate property value alias is guaranteed to constitute a unique string, as in “\p{bc=B}” versus “\p{ccc=B}”.

For a recommended use of property and property value aliases, see Unicode Technical Standard #18, “Unicode Regular Expressions.” Aliases are also used for normatively referencing properties, as described in *Section 3.1, Versions of the Unicode Standard*.

UCD in XML. Starting with Unicode Version 5.1.0, the complete Unicode Character Database is also available formatted in XML. This includes both the non-Han part of the Unicode Character Database and all of the content of the Unihan Database. For details regarding the XML schema, file names, grouping conventions, and other considerations, see Unicode Standard Annex #42, “Unicode Character Database in XML.”

Online Availability. All versions of the UCD are available online on the Unicode website. See the subsections “Online Unicode Character Database” and “Online Unihan Database” in *Section B.3, Other Unicode Online Resources.*

4.2 Case

Case is a normative property of characters in certain alphabets whereby characters are considered to be variants of a single letter. These variants, which may differ markedly in shape and size, are called the *uppercase* letter (also known as *capital* or *majuscule*) and the *lowercase* letter (also known as *small* or *minuscule*). The uppercase letter is generally larger than the lowercase letter.

Because of the inclusion of certain composite characters for compatibility, such as U+01F1 LATIN CAPITAL LETTER DZ, a third case, called *titlecase*, is used where the first character of a word must be capitalized. An example of such a character is U+01F2 LATIN CAPITAL LETTER D WITH SMALL LETTER Z. The three case forms are UPPERCASE, Titlecase, and lowercase.

For those scripts that have case (Latin, Greek, Coptic, Cyrillic, Glagolitic, Armenian, archaic Georgian, Deseret, and Warang Citi), uppercase characters typically contain the word *capital* in their names. Lowercase characters typically contain the word *small*. However, this is not a reliable guide. The word *small* in the names of characters from scripts other than those just listed has nothing to do with case. There are other exceptions as well, such as small capital letters that are not formally uppercase. Some Greek characters with *capital* in their names are actually titlecase. (Note that while the archaic Georgian script contained upper- and lowercase pairs, they are not used in modern Georgian. See *Section 7.7, Georgian*.)

Definitions of Case and Casing

The Unicode Standard has more than one formal definition of lowercase, uppercase, and related casing processes. This is the result of the inherent complexity of case relationships and of defining case-related behavior on the basis of individual character properties. This section clarifies the distinctions involved in the formal definition of casing in the standard. The additional complications for titlecase are omitted from the discussion; titlecase distinctions apply only to a handful of compatibility characters.

The first set of values involved in the definition of case are based on the General_Category property in UnicodeData.txt. The relevant values are General_Category = Ll (Lowercase_Letter) and General_Category = Lu (Uppercase_Letter). For most ordinary letters of bicameral scripts such as Latin, Greek, and Cyrillic, these values are obvious and non-problematical. However, the General_Category property is, by design, a partition of the Unicode codespace. This means that each Unicode character can only have one General_Category value, which results in some odd edge cases for modifier letters, letterlike symbols and letterlike numbers. As a consequence, not every Unicode character that *looks like* a lowercase character necessarily ends up with General_Category = Ll, and not every Unicode character that *looks like* an uppercase character ends up with General_Category = Lu.

The second set of definitions relevant to case consist of the derived binary properties, Lowercase and Uppercase, specified in DerivedCoreProperties.txt in the Unicode Character Database. Those derived properties augment the General_Category values by adding the additional characters that ordinary users think of as being lowercase or uppercase, based

primarily on their letterforms. The additional characters are included in the derivations by means of the contributory properties, Other_Lowercase and Other_Uppercase, defined in PropList.txt. For example, Other_Lowercase adds the various modifier letters that are letterlike in shape, the circled lowercase letter symbols, and the compatibility lowercase Roman numerals. Other_Uppercase adds the circled uppercase letter symbols, and the compatibility uppercase Roman numerals.

A third set of definitions for case is fundamentally different in kind, and does not consist of character properties at all. The functions isLowercase and isUppercase are string functions returning a binary True/False value. These functions are defined in *Section 3.13, Default Case Algorithms*, and depend on case mapping relations, rather than being based on letterforms per se. Basically, isLowercase is True for a string if the result of applying the toLowercase mapping operation for a string is the same as the string itself.

Table 4-1 illustrates the various possibilities for how these definitions interact, as applied to exemplary single characters or single character strings.

Table 4-1. Relationship of Casing Definitions

Code	Character	gc	Lowercase	Uppercase	isLowerCase(S)	isUpperCase(S)
0068	h	Ll	True	False	True	False
0048	H	Lu	False	True	False	True
24D7	ⓗ	So	True	False	True	False
24BD	Ⓗ	So	False	True	False	True
02B0	ʰ	Lm	True	False	True	True
1D34	ᴴ	Lm	True	False	True	True
02BD	ʽ	Lm	False	False	True	True

Note that for "caseless" characters, such as U+02B0, U+1D34, and U+02BD, isLowerCase and isUpperCase are *both* True, because the inclusion of a caseless letter in a string is not criterial for determining the casing of the string—a caseless letter always case maps to itself.

On the other hand, all modifier letters derived from letter shapes are also notionally lowercase, whether the letterform itself is a minuscule or a majuscule in shape. Thus U+1D34 modifier letter capital h is actually Lowercase = True. Other modifier letters not derived from letter shapes, such as U+02BD, are neither Lowercase nor Uppercase.

The string functions isLowerCase and isUpperCase also apply to strings longer than one character, of course, for which the character properties General_Category, LowerCase, and Uppercase are not relevant. In *Table 4-2*, the string function isTitleCase is also illustrated, to show its applicability for the same strings.

Programmers concerned with manipulating Unicode strings should generally be dealing with the string functions such as isLowerCase (and its functional cousin, toLowerCase), unless they are working directly with single character properties. Care is always advised, however, when dealing with case in the Unicode Standard, as expectations based simply on

Table 4-2. Case Function Values for Strings

Codes	String	isLowerCase(S)	isUpperCase(S)	isTitleCase(S)
0068 0068	hh	True	False	False
0048 0048	HH	False	True	False
0048 0068	Hh	False	False	True
0068 0048	hH	False	False	False

the behavior of the basic Latin alphabet (A..Z, a..z) do not generalize easily across the entire repertoire of Unicode characters, and because case for modifier letters, in particular, can result in unexpected behavior.

Case Mapping

The default case mapping tables defined in the Unicode Standard are normative, but may be overridden to match user or implementation requirements. The Unicode Character Database contains four files with case mapping information, as shown in *Table 4-3*. Full case mappings for Unicode characters are obtained by using the basic mappings from UnicodeData.txt and extending or overriding them where necessary with the mappings from SpecialCasing.txt. Full case mappings may depend on the context surrounding the character in the original string.

Some characters have a "best" single-character mapping in UnicodeData.txt as well as a full mapping in SpecialCasing.txt. Any character that does not have a mapping in these files is considered to map to itself. For more information on case mappings, see *Section 5.18, Case Mappings*.

Table 4-3. Sources for Case Mapping Information

File Name	Description
UnicodeData.txt	Contains the case mappings that map to a single character. These do not increase the length of strings, nor do they contain context-dependent mappings.
SpecialCasing.txt	Contains additional case mappings that map to more than one character, such as "ß" to "SS". Also contains context-dependent mappings, with flags to distinguish them from the normal mappings, as well as some locale-dependent mappings.
CaseFolding.txt	Contains data for performing locale-independent case folding, as described in "Caseless Matching," in *Section 5.18, Case Mappings*.
PropList.txt	Contains the definition of the property Soft_Dotted, which is used in the context specification for casing. See D138 in *Section 3.13, Default Case Algorithms*.

The single-character mappings in UnicodeData.txt are insufficient for languages such as German. Therefore, only legacy implementations that cannot handle case mappings that increase string lengths should use UnicodeData.txt case mappings alone.

A set of charts that show the latest case mappings is also available on the Unicode website. See "Charts" in *Section B.3, Other Unicode Online Resources*.

4.3 Combining Classes

Each combining character has a normative canonical *combining class*. This class is used with the Canonical Ordering Algorithm to determine which combining characters interact typographically and to determine how the canonical ordering of sequences of combining characters takes place. Class *zero* combining characters act like base letters for the purpose of determining canonical order. Combining characters with non-zero classes participate in reordering for the purpose of determining the canonical order of sequences of characters. (See *Section 3.11, Normalization Forms*, for the specification of the algorithm.)

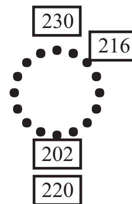
The list of combining characters and their canonical combining class appears in the Unicode Character Database. Most combining characters are nonspacing.

The canonical order of character sequences does *not* imply any kind of linguistic correctness or linguistic preference for ordering of combining marks in sequences. For more information on rendering combining marks, see *Section 5.13, Rendering Nonspacing Marks*.

Class zero combining marks are never reordered by the Canonical Ordering Algorithm. Except for class zero, the exact numerical values of the combining classes are of no importance in canonical equivalence, although the relative magnitude of the classes is significant. For example, it is crucial that the combining class of the cedilla be lower than the combining class of the dot below, although their exact values of 202 and 220 are not important for implementations.

Certain classes tend to correspond with particular rendering positions relative to the base character, as shown in *Figure 4-1*.

Figure 4-1. Positions of Common Combining Marks



Reordrant, Split, and Subjoined Combining Marks

In some scripts, the rendering of combining marks is notably complex. This is true in particular of the Brahmi-derived scripts of South and Southeast Asia, whose vowels are often encoded as class zero combining marks in the Unicode Standard, known as *matras* for the Indic scripts.

In the case of simple combining marks, as for the accent marks of the Latin script, the normative Unicode combining class of that combining mark typically corresponds to its posi-

tional placement with regard to a base letter, as described earlier. However, in the case of the combining marks representing vowels (and sometimes consonants) in the Brahmi-derived scripts and other *abugidas*, all of the combining marks are given the normative combining class of zero, regardless of their positional placement within an *aksara*. The placement and rendering of a class zero combining mark cannot be derived from its combining class alone, but rather depends on having more information about the particulars of the script involved. In some instances, the position may migrate in different historical periods for a script or may even differ depending on font style.

The identification of *matras* in Indic scripts is provided in the data file IndicSyllabicCategory.txt in the Unicode Character Database. Information about their positional placement can be found in the data file IndicPositionalCategory.txt. The following text in this section subcategorizes some of the class zero combining marks for Brahmi-derived scripts, pointing out significant types that need to be handled consistently, and relating their positional placement to the particular values documented in IndicPositionalCategory.txt.

Reordrant Class Zero Combining Marks. In many instances in Indic scripts, a vowel is represented in logical order *after* the consonant of a syllable, but is displayed *before* (to the left of) the consonant when rendered. Such combining marks are termed *reordrant* to reflect their visual reordering to the left of a consonant (or, in some instances, a consonant cluster). Special handling is required for selection and editing of these marks. In particular, the possibility that the combining mark may be reordered to the left side past a cluster, and not simply past the immediate preceding character in the backing store, requires attention to the details for each script involved.

The *visual* reordering of these reordrant class zero combining marks has nothing to do with the reordering of combining character sequences in the Canonical Ordering Algorithm. All of these marks are *class zero* and thus are *never* reordered by the Canonical Ordering Algorithm for normalization. The reordering is purely a presentational issue for *glyphs* during rendering of text.

Reordrant class zero combining marks correspond to the list of characters with Indic_Positional_Category = Left.

In addition, there are historically related vowel characters in the Thai, Lao, New Tai Lue, and Tai Viet scripts that are not treated as combining marks. Instead, for these scripts, such vowels are represented in the backing store in visual order and require no reordering for rendering. The trade-off is that they have to be rearranged for correct sorting. Because of that processing requirement, these characters are given a formal character property assignment, the Logical_Order_Exception property. See PropList.txt in the Unicode Character Database. The list of characters with the Logical_Order_Exception property is the same as those documented with the value Indic_Positional_Category = Visual_Order_Left in IndicPositionalCategory.txt.

Split Class Zero Combining Marks. In addition to the reordrant class zero combining marks, there are a number of class zero combining marks whose representative glyph typically consists of two parts, which are split into different positions with respect to the consonant (or consonant cluster) in an *aksara*. Sometimes these glyphic pieces are rendered

both to the left and the right of a consonant. Sometimes one piece is rendered above or below the consonant and the other piece is rendered to the left or the right. Particularly in the instances where some piece of the glyph is rendered to the left of the consonant, these split class zero combining marks pose similar implementation problems as for the reordrant marks.

The split class zero combining marks have various Indic_Positional_Category values such as Left_And_Right, Top_And_Bottom, Top_And_Right, Top_And_Left, and so forth. See IndicPositionalCategory.txt for the full listing.

One should pay very careful attention to all split class zero combining marks in implementations. Not only do they pose issues for rendering and editing, but they also often have canonical equivalences defined involving the separate pieces, when those pieces are also encoded as characters. As a consequence, the split combining marks may constitute exceptional cases under normalization. Some of the Tibetan split combining marks are deprecated.

The split vowels also pose difficult problems for understanding the standard, as the *phonological* status of the vowel phonemes, the *encoding* status of the characters (including any canonical equivalences), and the *graphical* status of the glyphs are easily confused, both for native users of the script and for engineers working on implementations of the standard.

Subjoined Class Zero Combining Marks. Brahmi-derived scripts that are not represented in the Unicode Standard with a virama may have class zero combining marks to represent subjoined forms of consonants. These correspond graphologically to what would be represented by a sequence of virama plus consonant in other related scripts. The subjoined consonants do not pose particular rendering problems, at least not in comparison to other combining marks, but they should be noted as constituting an exception to the normal pattern in Brahmi-derived scripts of consonants being represented with base letters. This exception needs to be taken into account when doing linguistic processing or searching and sorting.

Subjoined class zero combining marks are listed with the value Indic_Syllabic_Category = Consonant_Subjoined in IndicSyllabicCategory.txt.

Strikethrough Class Zero Combining Marks. The Kharoshthi script is unique in having some class zero combining marks for vowels that are struck through a consonant, rather than being placed in a position around the consonant. These strikethrough combining marks may involve particular problems for implementations. In addition to the Kharoshthi vowels, there are a number of combining *svarita* marks for Vedic texts which are also rendered as overstruck forms. These Kharoshthi vowels and Vedic *svarita* marks have the property value Indic_Positional_Category = Overstruck in IndicPositionalCategory.txt.

4.4 Directionality

Directional behavior is interpreted according to the Unicode Bidirectional Algorithm (see Unicode Standard Annex #9, "Unicode Bidirectional Algorithm"). For this purpose, all characters of the Unicode Standard possess a normative *directional* type, defined by the Bidi_Class (bc) property in the Unicode Character Database. The directional types left-to-right and right-to-left are called *strong types*, and characters of these types are called strong directional characters. Left-to-right types include most alphabetic and syllabic characters as well as all Han ideographic characters. Right-to-left types include the letters of predominantly right-to-left scripts, such as Arabic, Hebrew, and Syriac, as well as most punctuation specific to those scripts. In addition, the Unicode Bidirectional Algorithm uses *weak types* and *neutrals*. Interpretation of directional properties according to the Unicode Bidirectional Algorithm is needed for layout of right-to-left scripts such as Arabic and Hebrew.

4.5 General Category

The Unicode Character Database defines a General_Category property for all Unicode code points. The General_Category value for a character serves as a basic classification of that character, based on its primary usage. The property extends the widely used subdivision of ASCII characters into letters, digits, punctuation, and symbols—a useful classification that needs to be elaborated and further subdivided to remain appropriate for the larger and more comprehensive scope of the Unicode Standard.

Each Unicode code point is assigned a normative General_Category value. Each value of the General_Category is given a two-letter property value alias, where the first letter gives information about a major class and the second letter designates a subclass of that major class. In each class, the subclass "other" merely collects the remaining characters of the major class. For example, the subclass "No" (Number, other) includes all characters of the Number class that are not a decimal digit or letter. These characters may have little in common besides their membership in the same major class.

Table 4-4 enumerates the General_Category values, giving a short description of each value. See *Table 2-3* for the relationship between General_Category values and basic types of code points.

There are several other conventions for how General_Category values are assigned to Unicode characters. Many characters have multiple uses, and not all such uses can be captured by a single, simple partition property such as General_Category. Thus, many letters often serve dual functions as numerals in traditional numeral systems. Examples can be found in the Roman numeral system, in Greek usage of letters as numbers, in Hebrew, and similarly for many scripts. In such cases the General_Category is assigned based on the primary letter usage of the character, even though it may also have numeric values, occur in numeric expressions, or be used symbolically in mathematical expressions, and so on.

The General_Category gc=Nl is reserved primarily for letterlike number forms which are not technically digits. For example, the compatibility Roman numeral characters, U+2160..U+217F, all have gc=Nl. Because of the compatibility status of these characters, the recommended way to represent Roman numerals is with regular Latin letters (gc=Ll or gc=Lu). These letters derive their numeric status from conventional usage to express Roman numerals, rather than from their General_Category value.

Currency symbols (gc=Sc), by contrast, are given their General_Category value based entirely on their function as symbols for currency, even though they are often derived from letters and may appear similar to other diacritic-marked letters that get assigned one of the letter-related General_Category values.

Pairs of opening and closing punctuation are given their General_Category values (gc=Ps for opening and gc=Pe for closing) based on the most typical usage and orientation of such pairs. Occasional usage of such punctuation marks unpaired or in opposite orientation certainly occurs, however, and is in no way prevented by their General_Category values.

Table 4-4. General Category

Lu	=	Letter, uppercase
Ll	=	Letter, lowercase
Lt	=	Letter, titlecase
Lm	=	Letter, modifier
Lo	=	Letter, other
Mn	=	Mark, nonspacing
Mc	=	Mark, spacing combining
Me	=	Mark, enclosing
Nd	=	Number, decimal digit
Nl	=	Number, letter
No	=	Number, other
Pc	=	Punctuation, connector
Pd	=	Punctuation, dash
Ps	=	Punctuation, open
Pe	=	Punctuation, close
Pi	=	Punctuation, initial quote (may behave like Ps or Pe depending on usage)
Pf	=	Punctuation, final quote (may behave like Ps or Pe depending on usage)
Po	=	Punctuation, other
Sm	=	Symbol, math
Sc	=	Symbol, currency
Sk	=	Symbol, modifier
So	=	Symbol, other
Zs	=	Separator, space
Zl	=	Separator, line
Zp	=	Separator, paragraph
Cc	=	Other, control
Cf	=	Other, format
Cs	=	Other, surrogate
Co	=	Other, private use
Cn	=	Other, not assigned (including noncharacters)

Similarly, characters whose General_Category identifies them primarily as a symbol or as a mathematical symbol may function in other contexts as punctuation or even paired punctuation. The most obvious such case is for U+003C “<” LESS-THAN SIGN and U+003E “>” GREATER-THAN SIGN. These are given the General_Category gc = Sm because their primary identity is as mathematical relational signs. However, as is obvious from HTML and XML, they also serve ubiquitously as paired bracket punctuation characters in many formal syntaxes.

A common use of the General_Category of a Unicode character is in the derivation of properties for the determination of text boundaries, as in Unicode Standard Annex #29, “Unicode Text Segmentation.” Other common uses include determining language identifiers for programming, scripting, and markup, as in Unicode Standard Annex #31, “Unicode

Identifier and Pattern Syntax," and in regular expression languages such as Perl. For more information, see Unicode Technical Standard #18, "Unicode Regular Expressions."

This property is also used to support common APIs such as `isDigit()`. Common functions such as `isLetter()` and `isUppercase()` do not extend well to the larger and more complex repertoire of Unicode. While it is possible to naively extend these functions to Unicode using the General_Category and other properties, they will not work for the entire range of Unicode characters and the kinds of tasks for which people intend them. For more appropriate approaches, see Unicode Standard Annex #31, "Unicode Identifier and Pattern Syntax"; Unicode Standard Annex #29, "Unicode Text Segmentation"; *Section 5.18, Case Mappings*; and *Section 4.10, Letters, Alphabetic, and Ideographic*.

Although the General_Category property is normative, and its values are used in the derivation of many other properties referred to by Unicode algorithms, it does not follow that the General_Category always provides the most appropriate classification of a character for any given purpose. Implementations are not required to treat characters solely according to their General_Category values when classifying them in various contexts. The following examples illustrate some typical cases in which an implementation might reasonably diverge from General_Category values for a character when grouping characters as "punctuation," "symbols," and so forth.

- A character picker application might classify U+0023 # NUMBER SIGN among symbols, or perhaps under both symbols and punctuation.
- An "Ignore Punctuation" option for a search might choose not to ignore U+0040 @ COMMERCIAL AT.
- A layout engine might treat U+0021 ! EXCLAMATION MARK as a mathematical operator in the context of a mathematical equation, and lay it out differently than if the same character were used as terminal punctuation in text.
- A regular expression syntax could provide an operator to match all punctuation, but include characters other than those limited to gc = P (for example, U+00A7 § SECTION SIGN).

The general rule is that if an implementation *purports* to be using the Unicode General_Category property, then it must use the exact values specified in the Unicode Character Database for that claim to be conformant. Thus, if a regular expression syntax explicitly supports the Unicode General_Category property and matches gc = P, then that match must be based on the precise UCD values.

4.6 Numeric Value

Numeric_Value and Numeric_Type are normative properties of characters that represent *numbers*. Characters with a non-default Numeric_Type include numbers and number forms such as fractions, subscripts, superscripts, Roman numerals, encircled numbers, and many script-specific digits and numbers.

In some traditional numbering systems, ordinary letters may also be used with a numeric value. Examples include Greek letters used numerically, Hebrew gematria, and even Latin letters when used in outlines (II.A.1.b). Letter characters used in this way are not given Numeric_Type or Numeric_Value property values, to prevent simplistic parsers from treating them numerically by mistake. The Unicode Character Database gives the Numeric_Type and Numeric_Value property values only for Unicode characters that *normally* represent numbers.

Decimal Digits. Decimal digits, as commonly understood, are digits used to form decimal-radix numbers. They include script-specific digits, but exclude characters such as Roman numerals and Greek acrophonic numerals, which do not form decimal-radix expressions. (Note that <1, 5> = 15 = fifteen, but <I, V> = IV = four.)

The Numeric_Type = Decimal property value (which is correlated with the General_Category = Nd property value) is limited to those numeric characters that are used in decimal-radix numbers and for which a full set of digits has been encoded in a contiguous range, with ascending order of Numeric_Value, and with the digit zero as the first code point in the range.

Decimal digits, as defined in the Unicode Standard by these property assignments, exclude some characters, such as the CJK ideographic digits (see the first ten entries in *Table 4-5*), which are not encoded in a contiguous sequence. Decimal digits also exclude the compatibility subscript and superscript digits, to prevent simplistic parsers from misinterpreting their values in context. (For more information on superscript and subscripts, see *Section 22.4, Superscript and Subscript Symbols*.) Traditionally, the Unicode Character Database has given these sets of noncontiguous or compatibility digits the value Numeric_Type = Digit, to recognize the fact that they consist of digit values but do not necessarily meet all the criteria for Numeric_Type = Decimal. However, the distinction between Numeric_Type = Digit and the more generic Numeric_Type = Numeric has proven not to be useful in implementations. As a result, future sets of digits which may be added to the standard and which do not meet the criteria for Numeric_Type = Decimal will simply be assigned the value Numeric_Type = Numeric.

Numbers other than decimal digits can be used in numerical expressions, and may be interpreted by a numeric parser, but it is up to the implementation to determine such specialized uses.

Script-Specific Digits. The Unicode Standard encodes separate characters for the digits specific to a given script. Examples are the digits used with the Arabic script or those of the various Indic scripts. See *Table 22-3* for a list of script-specific digits. For naming conventions relevant to the Arabic digits, see the introduction to *Section 9.2, Arabic*.

Ideographic Numeric Values

CJK ideographs also may have numeric values. The primary numeric ideographs are shown in *Table 4-5*. When used to represent numbers in decimal notation, zero is represented by U+3007. Otherwise, zero is represented by U+96F6.

Table 4-5. Primary Numeric Ideographs

Code Point	**Value**
U+96F6	0
U+4E00	1
U+4E8C	2
U+4E09	3
U+56DB	4
U+4E94	5
U+516D	6
U+4E03	7
U+516B	8
U+4E5D	9
U+5341	10
U+767E	100
U+5343	1,000
U+4E07	10,000
U+5104	100,000,000 (10,000 × 10,000)
U+4EBF	100,000,000 (10,000 × 10,000)
U+5146	1,000,000,000,000 (10,000 × 10,000 × 10,000)

Ideographic accounting numbers are commonly used on checks and other financial instruments to minimize the possibilities of misinterpretation or fraud in the representation of numerical values. The set of accounting numbers varies somewhat between Japanese, Chinese, and Korean usage. *Table 4-6* gives a fairly complete listing of the known accounting characters. Some of these characters are ideographs with other meanings pressed into service as accounting numbers; others are used only as accounting numbers.

In Japan, U+67D2 is also pronounced *urusi*, meaning “lacquer,” and is treated as a variant of the standard character for “lacquer,” U+6F06.

The Unihan Database gives the most up-to-date and complete listing of primary numeric ideographs and ideographs used as accounting numbers, including those for CJK repertoire extensions beyond the Unified Repertoire and Ordering. See Unicode Standard Annex #38, “Unicode Han Database (Unihan),” for more details.

Table 4-6. Ideographs Used as Accounting Numbers

Number	Multiple Uses	Accounting Use Only
1	U+58F9, U+58F1	U+5F0C
2		U+8CAE, U+8CB3, U+8D30, U+5F10, U+5F0D
3	U+53C3, U+53C2	U+53C1, U+5F0E
4	U+8086	
5	U+4F0D	
6	U+9678, U+9646	
7	U+67D2	
8	U+634C	
9	U+7396	
10	U+62FE	
100	U+964C	U+4F70
1,000	U+4EDF	
10,000	U+842C	

4.7 Bidi Mirrored

Bidi Mirrored is a normative property of characters such as parentheses, whose images are mirrored horizontally in text that is laid out from right to left. For example, U+0028 LEFT PARENTHESIS is interpreted as *opening parenthesis*; in a left-to-right context it will appear as "(", while in a right-to-left context it will appear as the mirrored glyph ")". This requirement is necessary to render the character properly in a bidirectional context. Mirroring is the default behavior for such paired characters in Unicode text. (For more information, see the "Paired Punctuation" subsection in *Section 6.2, General Punctuation*.)

Paired delimiters are mirrored even when they are used in unusual ways, as, for example, in the mathematical expressions [a,b) or]a,b[. If any of these expression is displayed from right to left, then the mirrored glyphs are used. Because of the difficulty in interpreting such expressions, authors of bidirectional text need to make sure that readers can determine the desired directionality of the text from context.

Note that mirroring is not limited to paired punctuation and other paired delimiters, but also applies to a limited set of mathematical symbols whose orientation is reversed when the direction of line layout is reversed—for example, U+222B INTEGRAL. Such characters subject to bidi mirroring require the availability of a left-right symmetric pair of glyphs for correct display.

For some mathematical symbols, the "mirrored" form is not an exact mirror image. For example, the direction of the circular arrow in U+2232 CLOCKWISE CONTOUR INTEGRAL reflects the direction of the integration in coordinate space, not the text direction. In a right-to-left context, the integral sign would be mirrored, but the circular arrow would retain its direction. In a similar manner, the bidi-mirrored form of U+221B CUBE ROOT would be composed of a mirrored radix symbol with a non-mirrored digit "3". For more information, see Unicode Technical Report #25, "Unicode Support for Mathematics."

The list of mirrored characters appears in the Unicode Character Database. Formally, they consist of all characters with the property value Bidi_Mirrored = Y. This applies to almost all paired brackets (with the legacy exception of U+FD3E ORNATE LEFT PARENTHESIS and U+FD3F ORNATE RIGHT PARENTHESIS), but not to quotation marks, whose directionality and pairing status is less predictable than paired brackets. (See the subsection on "Language-Based Usage of Quotation Marks" in *Section 6.2, General Punctuation*.) Many mathematical operators with a directional orientation are bidi mirrored, but mirroring does *not* apply to any arrow symbols.

The mirroring behavior noted in paleographic materials for a number of ancient scripts, such as Old Italic, Runic, (ancient) Greek, Egyptian Hieroglyphs, and so forth, is *not* within the scope of the Bidi Mirrored property, and is not handled by default in the Unicode Bidirectional Algorithm (UBA). Mirroring of the letters or signs in the text of such paleographic material should be dealt with by higher level protocol. HL6 "Additional mirroring" is specified by the UBA as a permissible type of higher-level protocol to allow additional mirroring of glyphs for certain characters in a bidirectional context. A straightforward approach to a higher-level protocol would use existing bidirectional for-

mat controls to override text layout direction, add mirrored glyphs to a font used for paleographic display, and make the display choice depend on resolved direction for a directional run. HL3 "Emulate explicit directional formatting characters" in the UBA also allows a higher-level protocol to use other techniques such as style sheets or markup to override text directionality in structured text. In combination, such techniques can provide for the layout requirements of paleographic scripts which may mirror letters or signs depending on text layout direction. See the discussions of directionality and text layout in the respective sections regarding each script.

Related Properties. The *Bidi Mirrored* property is not to be confused with the related, informative *Bidi Mirroring Glyph* property, which lists pairs of characters whose representative glyphs are mirror images of each other. The Unicode Bidirectional Algorithm also requires two related, normative properties, *Bidi Paired Bracket* and *Bidi Paired Bracket Type*, which are used for matching specific bracket pairs and to assign the same text direction to both members of each pair in bidirectional processing for text layout. These properties do not affect mirroring. For more information, see BidiMirroring.txt and BidiBrackets.txt in the Unicode Character Database.

4.8 Name

Unicode characters have names that serve as unique identifiers for each character. The character names in the Unicode Standard are identical to those of the English-language edition of ISO/IEC 10646.

Where possible, character names are derived from existing conventional names of a character or symbol in English, but in many cases the character names nevertheless differ from traditional names widely used by relevant user communities. The character names of symbols and punctuation characters often describe their shape, rather than their function, because these characters are used in many different contexts. See also "Color Words in Unicode Character Names" in *Section 22.9, Miscellaneous Symbols*.

Character names are listed in the code charts. Currently, the character with the longest name is U+FBF9 ARABIC LIGATURE UIGHUR KIRGHIZ YEH WITH HAMZA ABOVE WITH ALEF MAKSURA ISOLATED FORM (Version 1.1) with 83 letters and spaces in its name, and the one with the shortest name is U+1F402 OX (Version 6.0) with only two letters in its name.

Stability. Once assigned, a character name is immutable. It will never be changed in subsequent versions of the Unicode Standard. Implementers and users can rely on the fact that a character name uniquely represents a given character.

Character Name Syntax. Unicode character names, as listed in the code charts, contain only uppercase Latin letters A through Z, digits, space, and hyphen-minus. In more detail, character names reflect the following rules:

R1 ***Only Latin capital letters A to Z (U+0041..U+005A), ASCII digits (U+0030.. U+0039), U+0020*** SPACE, ***and U+002D*** HYPHEN-MINUS ***occur in character names.***

R2 ***Digits do not occur as the first character of a character name, nor immediately following a space character.***

R3 ***U+002D*** HYPHEN-MINUS ***does not occur as the first or last character of a character name, nor immediately between two spaces, nor immediately preceding or following another hyphen-minus character. (In other words, multiple occurrences of U+002D in sequence are not allowed.)***

R4 ***A space does not occur as the first or last character of a character name, nor immediately preceding or following another space character. (In other words, multiple spaces in sequence are not allowed.)***

See *Appendix A, Notational Conventions*, for the typographical conventions used when printing character names in the text of the standard.

Names as Identifiers. Character names are constructed so that they can easily be transposed into formal identifiers in another context, such as a computer language. Because Unicode character names do not contain any underscore ("_") characters, a common strategy is to replace any *hyphen-minus* or space in a character name by a single "_" when constructing a formal identifier from a character name. This strategy automatically results in a

syntactically correct identifier in most formal languages. Furthermore, such identifiers are guaranteed to be unique, because of the special rules for character name matching.

Character Name Matching. When matching identifiers transposed from character names, it is possible to ignore case, whitespace, and all medial *hyphen-minus* characters (or any "_" replacing a *hyphen-minus*), except for the *hyphen-minus* in U+1180 HANGUL JUNGSEONG O-E, and still result in a unique match. For example, "ZERO WIDTH SPACE" is equivalent to "zero-width-space" or "ZERO_WIDTH_SPACE" or "ZeroWidthSpace". However, "TIBETAN LETTER A" should not match "TIBETAN LETTER -A", because in that instance the *hyphen-minus* is not medial between two letters, but is instead preceded by a space. For more information on character name matching, see Section 5.9, "Matching Rules" in Unicode Standard Annex #44, "Unicode Character Database."

Named Character Sequences. Occasionally, character sequences are also given a normative name in the Unicode Standard. The names for such sequences are taken from the same namespace as character names, and are also unique. For details, see Unicode Standard Annex #34, "Unicode Named Character Sequences." Named character sequences are not listed in the code charts; instead, they are listed in the file NamedSequences.txt in the Unicode Character Database.

The names for named character sequences are also immutable. Once assigned, they will never be changed in subsequent versions of the Unicode Standard.

Character Name Aliases. The Unicode Standard has a mechanism for the publication of additional, normative formal aliases for characters. These formal aliases are known as *character name aliases.* (See Definition D5 in *Section 3.3, Semantics.*) They function essentially as auxiliary names for a character. The original reason for defining character name aliases was to provide corrections for known mistakes in character names, but they have also proven useful for other purposes, as documented here.

Character name aliases are listed in the file NameAliases.txt in the Unicode Character Database. That file also documents the type field which distinguishes among different kinds of character name aliases, as shown in *Table 4-7*.

Table 4-7. Types of Character Name Aliases

Type	Description
correction	Corrections for serious problems in the character names
control	ISO 6429 names for C0 and C1 control functions, and other commonly occurring names for control codes
alternate	Widely used alternate names for format characters
figment	Several documented labels for C1 control code points which were never actually approved in any standard
abbreviation	Commonly occurring abbreviations (or acronyms) for control codes, format characters, spaces, and variation selectors

Character name aliases are immutable, once published. (See Definition D42 in *Section 3.5, Properties.*) They follow the same syntax rules as character names and are also guaranteed

to be unique in the Unicode namespace for character names. This attribute makes character name aliases useful as identifiers. A character may, in principle, have more than one normative character name alias, but each distinct character name alias uniquely identifies only a single code point.

The first type of character name alias consists of corrections for known mistakes in character names. Sometimes errors in a character name are only discovered after publication of a version of the Unicode Standard. Because character names are immutable, such errors are not corrected by changing the names after publication. However, in some limited instances (as for obvious typos in the name), a character name alias is defined instead.

For example, the following Unicode character name has a well-known spelling error in it:

U+FE18 PRESENTATION FORM FOR VERTICAL RIGHT WHITE LENTICULAR BRAKCET

Because the spelling error could not be corrected after publication of the data files which first contained it, a character name alias with the corrected spelling was defined:

U+FE18 PRESENTATION FORM FOR VERTICAL RIGHT WHITE LENTICULAR BRACKET

Character name aliases are provided for additional reasons besides corrections of errors in the character names. For example, there are character name aliases which give definitive labels to control codes, which have no actual Unicode character names:

U+0009 HORIZONTAL TABULATION

Character name aliases of type *alternate* are for widely used alternate names of Unicode format characters. Currently only one such alternate is normatively defined, but it is for an important character:

U+FEFF BYTE ORDER MARK

Among the control codes there are a few which have had names propagate through the computer implementation “lore,” despite the fact that they refer to ISO/IEC 10646 control functions that were never formally adopted. These names are defined as character name aliases of type *figment*, and are included in NameAliases.txt, because they occur in some widely distributed implementations, such as the regex engine for Perl. Examples include:

U+0081 HIGH OCTET PRESET

Additional character name aliases match existing and widely used abbreviations (or acronyms) for control codes and for Unicode format characters:

U+0009 TAB

U+200B ZWSP

Specifying these additional, normative character name aliases serves two major functions. First, it provides a set of well-defined aliases for use in regular expression matching and searching, where users might expect to be able to use established names or abbreviations for control codes and the like, but where those names or abbreviations are not part of the actual Unicode Name property. Second, because character name aliases are guaranteed to

be unique in the Unicode character name namespace, having them defined for control codes and abbreviations prevents the potential for accidental collisions between de facto current use and names which might be chosen in the future for newly encoded Unicode characters.

It is acceptable and expected for external specifications to make normative references to Unicode characters using one (or more) of their normative character name aliases, where such references make sense. For example, when discussing Unicode encoding schemes and the role of U+FEFF as a signature for byte order, it would not make much sense to insist on referring to U+FEFF by its name ZERO WIDTH NO-BREAK SPACE, when use of the character name alias BYTE ORDER MARK or the widely used abbreviation BOM would communicate with less confusion.

A subset of character name aliases is listed in the code charts, using special typographical conventions explained in *Section 24.1, Character Names List.*

A normative character name alias is distinct from the informative aliases listed in the code charts. Informative aliases merely point out other common names in use for a given character. Informative aliases are not immutable and are not guaranteed to be unique; they therefore cannot serve as an identifier for a character. Their main purposes are to help readers of the standard to locate and to identify particular characters.

Unicode Name Property

Formally, the character name for a Unicode character is the value of the normative character property, "Name". Most Unicode character properties are defined by enumeration in one of the data files of the Unicode Character Database, but the Name property is instead defined in part by enumeration and in part by rule. A significant proportion of Unicode characters belong to large sets, such as Han ideographs, Tangut ideographs, and Hangul syllables, for which the character names are best defined by generative rule, rather than one-by-one naming.

Formal Definition of the Name Property. The Name property (short alias: "na") is a string property, defined as follows:

NR1 ***For Hangul syllables, the Name property value is derived by rule, as specified in Section 3.12, Conjoining Jamo Behavior, under "Hangul Syllable Name Generation," by concatenating a fixed prefix string "HANGUL SYLLABLE " and appropriate values of the Jamo_Short_Name property.***

For example, the name of U+D4DB is HANGUL SYLLABLE PWILH, constructed by concatenation of "HANGUL SYLLABLE " and three Jamo_Short_Name property values, "P", + "WI" + "LH".

NR2 ***For most ideographs (characters with the binary property value Ideographic = True), the Name property value is derived by concatenating a script-specific prefix string, as specified in Table 4-8, to the code point, expressed in hexadecimal, with the usual 4- to 6-digit convention.***

For example, the name of U+4E00 is CJK UNIFIED IDEOGRAPH-4E00, constructed by concatenation of "CJK UNIFIED IDEOGRAPH-" and the code point. Similarly, the character name of U+17000 is TANGUT IDEOGRAPH-17000.

NR3 For all other Graphic characters and for all Format characters, the Name property value is as explicitly listed in Field 1 of UnicodeData.txt.

For example, U+0A15 GURMUKHI LETTER KA or U+200D ZERO WIDTH JOINER.

NR4 For all* other *Unicode code points of all other types (Control, Private-Use, Surrogate, Noncharacter, and Reserved), the value of the Name property is the null string. In other words, na = "".

The ranges of Hangul syllables and most ideographic characters subject to the name derivation rules NR1 and NR2 are identified by a special convention in Field 1 of UnicodeData.txt. The start and end of each range are indicated by a pair of entries in the data file in the general format:

```
NNNN;<RANGENAME, First>;Lo;0;L;;;;;N;;;;;
NNNN;<RANGENAME, Last>;Lo;0;L;;;;;N;;;;;
```

This convention originated as a compression technique for UnicodeData.txt, as all of the UnicodeData.txt properties of these ranges were uniform, and the names for the characters in the ranges could be specified by rule. Note that the same convention is used in UnicodeData.txt to specify properties for code point types which have a null string as their Name property value, such as private use characters.

CJK compatibility ideographs are an exception. They have names derived by rule NR2, but are *explicitly* listed in UnicodeData.txt with their names, because they typically have nonuniform character properties, including most notably a nontrivial canonical decomposition value.

The exact ranges subject to name derivation rules NR1 and NR2, and the specified prefix strings are summarized in *Table 4-8*.

Twelve of the CJK ideographs in the starred range in *Table 4-8*, in the CJK Compatibility Ideographs block, are actually CJK unified ideographs. Nonetheless, their names are constructed with the "CJK COMPATIBILITY IDEOGRAPH-" prefix shared by all other code points in that block. The status of a CJK ideograph as a unified ideograph cannot be deduced from the Name property value for that ideograph; instead, the dedicated binary property Unified_Ideograph should be used to determine that status. See "CJK Compatibility Ideographs" in *Section 18.1, Han*, and Section 4.4, "Listing of Characters Covered by the Unihan Database" in Unicode Standard Annex #38, "Unihan Database," for more details about these exceptional twelve CJK ideographs.

The generic term "character name" refers to the Name property value for an encoded Unicode character. An expression such as, "The reserved code point U+30000 has no name," is shorthand for the more precise statement that the reserved code point U+30000 (as for all code points of type Reserved) has a property value of na = "" for the Name property.

Table 4-8. Name Derivation Rule Prefix Strings

Range	Rule	Prefix String
AC00..D7A3	NR1	"HANGUL SYLLABLE"
3400..4DB5	NR2	"CJK UNIFIED IDEOGRAPH-"
4E00..9FEA	NR2	"CJK UNIFIED IDEOGRAPH-"
20000..2A6D6	NR2	"CJK UNIFIED IDEOGRAPH-"
2A700..2B734	NR2	"CJK UNIFIED IDEOGRAPH-"
2B740..2B81D	NR2	"CJK UNIFIED IDEOGRAPH-"
2B820..2CEA1	NR2	"CJK UNIFIED IDEOGRAPH-"
2CEB0..2EBE0	NR2	"CJK UNIFIED IDEOGRAPH-"
17000..187EC	NR2	"TANGUT IDEOGRAPH-"
1B170..1B2FB	NR2	"NUSHU CHARACTER-"
F900..FA6D*	NR2	"CJK COMPATIBILITY IDEOGRAPH-"
FA70..FAD9	NR2	"CJK COMPATIBILITY IDEOGRAPH-"
2F800..2FA1D	NR2	"CJK COMPATIBILITY IDEOGRAPH-"

Name Uniqueness. The Unicode Name property values are unique for all non-null values, but not every Unicode code point has a unique Unicode Name property value. Furthermore, because Unicode character names, character name aliases, and named character sequences constitute a single, unique namespace, the Name property value uniqueness requirement applies to all three kinds of names.

Interpretation of Field 1 of UnicodeData.txt. Where Field 1 of UnicodeData.txt contains a string enclosed in angle brackets, "<" and ">", such a string is *not* a character name, but a meta-label indicating some other information—for example, the start or end of a character range. In these cases, the Name property value for that code point is either empty (na = "") or is given by one of the rules described above. In all *other* cases, the value of Field 1 (that is, the string of characters between the first and second semicolon separators on each line) corresponds to the normative value of the Name property for that code point.

Control Codes. The Unicode Standard does not define character names for control codes (characters with General_Category = Cc). In other words, all control codes have a property value of na = "" for the Name property. Control codes are instead listed in UnicodeData.txt with a special label "<control>" in Field 1. This value is not a character name, but instead indicates the *code point type* (see Definition D10a in *Section 3.4, Characters and Encoding*). For control characters, the values of the *informative* Unicode 1.0 name property (Unicode_1_Name) in Field 10 match the names of the associated control functions from ISO/IEC 6429. (See *Section 4.9, Unicode 1.0 Names.*)

Code Point Labels

To provide unique, meaningful labels for code points that do not have character names, the Unicode Standard uses a convention for code point labeling.

For each code point type without character names, code point labels are constructed by using a lowercase prefix derived from the code point type, followed by a *hyphen-minus* and then a 4- to 6-digit hexadecimal representation of the code point. The label construction for the five affected code point types is illustrated in *Table 4-9*.

Table 4-9. Construction of Code Point Labels

Type	Label
Control	control-NNNN
Reserved	reserved-NNNN
Noncharacter	noncharacter-NNNN
Private-Use	private-use-NNNN
Surrogate	surrogate-NNNN

To avoid any possible confusion with actual, non-null Name property values, constructed Unicode code point labels are often displayed between angle brackets: <control-0009>, <noncharacter-FFFF>, and so on. This convention is used consistently in the data files for the Unicode Character Database.

A constructed code point label is distinguished from the designation of the code point itself (for example, "U+0009" or "U+FFFF"), which is also a unique identifier, as described in *Appendix A, Notational Conventions*.

Use of Character Names in APIs and User Interfaces

Use in APIs. APIs which return the value of a Unicode "character name" for a given code point might vary somewhat in their behavior. An API which is defined as *strictly* returning the value of the Unicode Name property (the "na" attribute), should return a null string for any Unicode code point other than graphic or format characters, as that is the actual value of the property for such code points. On the other hand, an API which returns a name for Unicode code points, but which is expected to provide useful, unique labels for unassigned, reserved code points and other special code point types, should return the value of the Unicode Name property for any code point for which it is non-null, but should otherwise construct a code point label to stand in for a character name.

User Interfaces. A list of Unicode character names may not always be the most appropriate set of choices to present to a user in a user interface. Many common characters do not have a single name for all English-speaking user communities and, of course, their native name in another language is likely to be different altogether. The names of many characters in the Unicode Standard are based on specific Latin transcription of the sounds they represent. There are often competing transcription schemes. For all these reasons, it can be more effective for a user interface to use names that were translated or otherwise adjusted to meet the expectations of the targeted user community. By also listing the formal character name, a user interface could ensure that users can unambiguously refer to the character by the name documented in the Unicode Standard.

4.9 Unicode 1.0 Names

The Unicode_1_Name property is an informative property referring to the name of characters in Version 1.0 of the Unicode Standard. Values of the Unicode_1_Name property are provided in UnicodeData.txt in the Unicode Character Database in cases where the Version 1.0 name of a character differed from the current name of that character. A significant number of names for Unicode characters in Version 1.0 were changed during the process of merging the repertoire of the Unicode Standard with ISO/IEC 10646 in 1991. Character name changes are now strictly prohibited by the Unicode Character Encoding Stability Policy, and no character name has been changed since Version 2.0.

The Version 1.0 names are primarily of historic interest regarding the early development of the Unicode Standard. However, where a Version 1.0 character name provides additional useful information about the identity of a character, it is explicitly listed in the code charts. For example, U+00B6 PILCROW SIGN has its Version 1.0 name, PARAGRAPH SIGN, listed for clarity.

The status of the Unicode_1_Name property values in the case of control codes differs from that for other characters. *The Unicode Standard, Version 1.0,* gave names to the C0 control codes, U+0000..U+001F, U+007F, based on then-current practice for reference to ASCII control codes. Unicode 1.0 gave no names to the C1 control codes, U+0080..U+009F. The values of the Unicode_1_Name property have been updated for the control codes to reflect the ISO/IEC 6429 standard names for control functions. Those names can be seen as annotations in the code charts. In a few instances, because of updates to ISO/IEC 6429, those names may differ from the names that actually occurred in Unicode 1.0. For example, the Unicode 1.0 name of U+0009 was HORIZONTAL TABULATION, but the ISO/IEC 6429 name for this function is CHARACTER TABULATION, and the commonly used alias is, of course, merely *tab*.

4.10 Letters, Alphabetic, and Ideographic

Letters and Syllables. The concept of a letter is used in many contexts. Computer language standards often characterize identifiers as consisting of letters, syllables, ideographs, and digits, but do not specify exactly what a "letter," "syllable," "ideograph," or "digit" is, leaving the definitions implicitly either to a character encoding standard or to a locale specification. The large scope of the Unicode Standard means that it includes many writing systems for which these distinctions are not as self-evident as they may once have been for systems designed to work primarily for Western European languages and Japanese. In particular, while the Unicode Standard includes various "alphabets" and "syllabaries," it also includes writing systems that fall somewhere in between. As a result, no attempt is made to draw a sharp property distinction between letters and syllables.

Alphabetic. The Alphabetic property is a derived informative property of the primary units of alphabets and/or syllabaries, whether combining or noncombining. Included in this group would be composite characters that are canonical equivalents to a combining character sequence of an alphabetic base character plus one or more combining characters; letter digraphs; contextual variants of alphabetic characters; ligatures of alphabetic characters; contextual variants of ligatures; modifier letters; letterlike symbols that are compatibility equivalents of single alphabetic letters; and miscellaneous letter elements. Notably, U+00AA FEMININE ORDINAL INDICATOR and U+00BA MASCULINE ORDINAL INDICATOR are simply abbreviatory forms involving a Latin letter and should be considered alphabetic rather than nonalphabetic symbols.

Ideographic. The Ideographic property is an informative property defined in the Unicode Character Database. The Ideographic property is used, for example, in determining line breaking behavior. Characters with the Ideographic property include unified CJK ideographs, CJK compatibility ideographs, Tangut ideographs, Nüshu ideographs, and characters from other blocks—for example, U+3007 IDEOGRAPHIC NUMBER ZERO and U+3006 IDEOGRAPHIC CLOSING MARK. For more information about Han, Tangut, and Nüshu ideographs, see *Section 18.1, Han*, *Section 18.11, Tangut* and *Section 18.8, Nüshu*. For more about ideographs and logosyllabaries in general, see *Section 6.1, Writing Systems*.

4.11 Properties for Text Boundaries

The determination of text boundaries, such as word breaks or line breaks, involves contextual analysis of potential break points and the characters that surround them. Such an analysis is based on the classification of all Unicode characters by their default interaction with each particular type of text boundary. For example, the Line_Break property defines the default behavior of Unicode characters with respect to line breaking.

A number of characters have special behavior in the context of determining text boundaries. These characters are described in more detail in the subsection on "Line and Word Breaking" in *Section 23.2, Layout Controls*. For more information about text boundaries and these characters, see Unicode Standard Annex #14, "Unicode Line Breaking Algorithm," and Unicode Standard Annex #29, "Unicode Text Segmentation."

4.12 Characters with Unusual Properties

The behavior of most characters does not require special attention in this standard. However, the characters in *Table 4-10* exhibit special behavior. Many other characters behave in special ways but are not noted here, either because they do not affect surrounding text in the same way or because their use is intended for well-defined contexts. Examples include the compatibility characters for block drawing, the symbol pieces for large mathematical operators, and many punctuation symbols that need special handling in certain circumstances. Such characters are more fully described in the following chapters. The section numbers or other references listed in the “Details” column in *Table 4-10* indicate where to find more information about the functions or particular groups of characters listed.

Table 4-10. Unusual Properties

Function	Details	Code Point and Name
Segmentation		
Line break controls	*Section 23.2*	00AD SOFT HYPHEN 200B ZERO WIDTH SPACE 2060 WORD JOINER
Combining Marks		
Bases for display of isolated nonspacing marks	*Section 2.11*, *Section 6.2*, *Section 23.2*	0020 SPACE 00A0 NO-BREAK SPACE
Double nonspacing marks	*Section 7.9*	035C COMBINING DOUBLE BREVE BELOW 035D COMBINING DOUBLE BREVE 035E COMBINING DOUBLE MACRON 035F COMBINING DOUBLE MACRON BELOW 0360 COMBINING DOUBLE TILDE 0361 COMBINING DOUBLE INVERTED BREVE 0362 COMBINING DOUBLE RIGHTWARDS ARROW BELOW 1DCD COMBINING DOUBLE CIRCUMFLEX ABOVE 1DFC COMBINING DOUBLE INVERTED BREVE BELOW
Combining half marks	*Section 7.9*	FE20 COMBINING LIGATURE LEFT HALF FE21 COMBINING LIGATURE RIGHT HALF and all other pairs in the Combining Half Marks block
Combining continuous lining marks	*Section 7.3*, *Section 7.9*	0305 COMBINING OVERLINE 0332 COMBINING LOW LINE 0333 COMBINING DOUBLE LOW LINE 033F COMBINING DOUBLE OVERLINE FE26 COMBINING CONJOINING MACRON FE2D COMBINING CONJOINING MACRON BELOW
Combining marks with non-default stacking	*Section 7.9*	1ABB COMBINING PARENTHESES ABOVE 1ABC COMBINING DOUBLE PARENTHESES ABOVE 1ABD COMBINING PARENTHESES BELOW

Table 4-10. Unusual Properties (Continued)

Function	Details	Code Point and Name
Ligation		
Cursive joining and ligation control	*Section 23.2*	200C ZERO WIDTH NON-JOINER 200D ZERO WIDTH JOINER
Fraction formatting	*Section 6.2*	2044 FRACTION SLASH
Ligating modifier tone letters	*Section 7.8*	02E5..02E9 MODIFIER LETTER EXTRA-HIGH TONE BAR..MODIFIER LETTER EXTRA-LOW TONE BAR A712..A716 MODIFIER LETTER EXTRA-HIGH LEFT-STEM TONE BAR..MODIFIER LETTER EXTRA-LOW LEFT-STEM TONE BAR
Ligating brackets that surround text	*Section 11.4, Section 13.4,*	0F3C TIBETAN MARK ANG KHANG GYON 0F3D TIBETAN MARK ANG KHANG GYAS 13258..1325D EGYPTIAN HIEROGLYPH O006A..EGYPTIAN HIEROGLYPH O006F 13282 EGYPTIAN HIEROGLYPH O033A 13286..13289 EGYPTIAN HIEROGLYPH O036A..EGYPTIAN HIEROGLYPH O036D 13379..1337B EGYPTIAN HIEROGLYPH V011A..EGYPTIAN HIEROGLYPH V011C
Ligating regional indicator symbols	*Section 22.10,* UTS #51	1F1E6..1F1FF REGIONAL INDICATOR SYMBOL LETTER A..REGIONAL INDICATOR SYMBOL LETTER Z
Indic-related: conjuncts, killers, and other viramas		
Brahmi-derived script dead-character formation	*Chapter 12, Chapter 13, Chapter 14, Chapter 15, Chapter 16*	See IndicSyllabicCategory.txt in the UCD for a full listing.
Brahmi number formation	*Section 14.1*	1107F BRAHMI NUMBER JOINER
Non-Indic consonant ligation	*Section 19.3*	2D7F TIFINAGH CONSONANT JOINER
Historical viramas with other functions	*Section 13.4, Section 13.6, Section 13.7, Section 13.11, Section 16.3*	0F84 TIBETAN MARK HALANTA 103A MYANMAR SIGN ASAT 193B LIMBU SIGN SA-I ABED MEETEI MAYEK APUN IYEK 11134 CHAKMA MAAYYAA
Ideographic-related		
Ideographic variation indication	*Section 6.2*	303E IDEOGRAPHIC VARIATION INDICATOR
Ideographic description	*Section 18.2*	2FF0..2FFB IDEOGRAPHIC DESCRIPTION CHARACTER LEFT TO RIGHT..IDEOGRAPHIC DESCRIPTION CHARACTER OVERLAID

Table 4-10. Unusual Properties (Continued)

Function	Details	Code Point and Name
Complex expression format control (scoped)		
Bidirectional ordering	*Section 23.2*	See *Table 23-3* for a full listing.
Mathematical expression processing and formatting	*Section 22.6*	2061 FUNCTION APPLICATION 2062 INVISIBLE TIMES 2063 INVISIBLE SEPARATOR 2064 INVISIBLE PLUS
Musical format control	*Section 21.2*	1D173 MUSICAL SYMBOL BEGIN BEAM 1D174 MUSICAL SYMBOL END BEAM 1D175 MUSICAL SYMBOL BEGIN TIE 1D176 MUSICAL SYMBOL END TIE 1D177 MUSICAL SYMBOL BEGIN SLUR 1D178 MUSICAL SYMBOL END SLUR 1D179 MUSICAL SYMBOL BEGIN PHRASE 1D17A MUSICAL SYMBOL END PHRASE
Prefixed format control	*Section 9.2, Section 9.3, Section 15.2*	0600 ARABIC NUMBER SIGN 0601 ARABIC SIGN SANAH 0602 ARABIC FOOTNOTE MARKER 0603 ARABIC SIGN SAFHA 0604 ARABIC SIGN SAMVAT 0605 ARABIC NUMBER MARK ABOVE 06DD ARABIC END OF AYAH 070F SYRIAC ABBREVIATION MARK 08E2 ARABIC DISPUTED END OF AYAH 110BD KAITHI NUMBER SIGN 110CD KAITHI NUMBER SIGN ABOVE
Interlinear annotation	*Section 23.8*	FFF9 INTERLINEAR ANNOTATION ANCHOR FFFA INTERLINEAR ANNOTATION SEPARATOR FFFB INTERLINEAR ANNOTATION TERMINATOR
Deprecated alternate formatting	*Section 23.3*	206A INHIBIT SYMMETRIC SWAPPING 206B ACTIVATE SYMMETRIC SWAPPING 206C INHIBIT ARABIC FORM SHAPING 206D ACTIVATE ARABIC FORM SHAPING 206E NATIONAL DIGIT SHAPES 206F NOMINAL DIGIT SHAPES
Other unusual format control		
Miao tonal vowel position control	*Section 18.10*	16F8F MIAO TONE RIGHT 16F90 MIAO TONE TOP RIGHT 16F91 MIAO TONE ABOVE 16F92 MIAO TONE BELOW
Shorthand format control	*Section 21.5*	1BC9D DUPLOYAN THICK LETTER SELECTOR 1BCA0 SHORTHAND FORMAT LETTER OVERLAP 1BCA1 SHORTHAND FORMAT CONTINUING OVERLAP 1BCA2 SHORTHAND FORMAT DOWN STEP 1BCA3 SHORTHAND FORMAT UP STEP
SignWriting fill and rotation	*Section 21.6*	1DA9B..1DA9F SIGNWRITING FILL MODIFIER-2..SIGNWRITING FILL MODIFIER-6 1DAA1..1DAAF SIGNWRITING ROTATION MODIFIER-2..SIGNWRITING ROTATION MODIFIER-16

Table 4-10. Unusual Properties (Continued)

Function	Details	Code Point and Name
Variation selection		
Generic variation selectors	*Section 23.4*	FE00..FE0F VARIATION SELECTOR-1..VARIATION SELECTOR-16 E0100..E01EF VARIATION SELECTOR-17..VARIATION SELECTOR-256
Mongolian variation selectors	*Section 13.5*	180B MONGOLIAN FREE VARIATION SELECTOR ONE 180C MONGOLIAN FREE VARIATION SELECTOR TWO 180D MONGOLIAN FREE VARIATION SELECTOR THREE 180E MONGOLIAN VOWEL SEPARATOR
Emoji modifiers for skin tone	*Section 22.9*, UTS #51	1F3FB..1F3FF EMOJI MODIFIER FITZPATRICK TYPE-1-2..EMOJI MODIFIER FITZPATRICK TYPE-6
Emoji components to indicate hair style	UTS #51	1F9B0..1F9B3 EMOJI COMPONENT RED HAIR..EMOJI COMPONENT WHITE HAIR
Tag characters		
Deprecated language tag	*Section 23.9*	E0001 LANGUAGE TAG
Tag characters	*Section 23.9*	E0020..E007F TAG SPACE..CANCEL TAG
Miscellaneous		
Collation weighting and sequence interpretation	*Section 23.2*	034F COMBINING GRAPHEME JOINER
Byte order signature	*Section 23.8*	FEFF ZERO WIDTH NO-BREAK SPACE
Object replacement	*Section 23.8*	FFFC OBJECT REPLACEMENT CHARACTER
Code conversion fallback	*Section 23.8*	FFFD REPLACEMENT CHARACTER