L2/24-189

Technical Reports

Unicode® Standard Annex #29

UNICODE TEXT SEGMENTATION

Version	Unicode 16.0.0 (draft)
Editors	Josh Hadley (johadley@adobe.com)
Date	2024-06-06
This Version	https://www.unicode.org/reports/tr29/tr29-44.html
Previous Version	https://www.unicode.org/reports/tr29/tr29-43.html
Latest Version	https://www.unicode.org/reports/tr29/
Latest Proposed Update	https://www.unicode.org/reports/tr29/proposed.html
Revision	44

Summary

This annex describes guidelines for determining default segmentation boundaries between certain significant text elements: grapheme clusters ("user-perceived characters"), words, and sentences. For line boundaries, see [UAX14] .

Status

*This is a **draft** document which may be updated, replaced, or superseded by other documents at any time. Publication does not imply endorsement by the Unicode Consortium. This is not a stable document; it is inappropriate to cite this document as other than a work in progress.*

> ***A Unicode Standard Annex (UAX)** forms an integral part of the Unicode Standard, but is published online as a separate document. The Unicode Standard may require conformance to normative content in a Unicode Standard Annex, if so specified in the Conformance chapter of that version of the Unicode Standard. The version number of a UAX document corresponds to the version of the Unicode Standard of which it forms a part.*

Please submit corrigenda and other comments with the online reporting form [Feedback]. Related information that is useful in understanding this annex is found in Unicode Standard Annex #41, "Common References for Unicode Standard Annexes." For the latest version of the Unicode Standard, see [Unicode]. For a list of current Unicode Technical Reports, see [Reports]. For more information about versions of the Unicode Standard, see [Versions]. For any errata which may apply to this annex, see [Errata].

Contents

1 Introduction

This annex describes guidelines for determining default boundaries between certain significant text elements: user-perceived characters, words, and sentences. The process of boundary determination is also called *segmentation*.

A string of Unicode-encoded text often needs to be broken up into text elements programmatically. Common examples of text elements include what users think of as characters, words, lines (more precisely, where line breaks are allowed), and sentences. The precise determination of text elements may vary according to orthographic conventions for a given script or language. The goal of matching user perceptions cannot always be met exactly because the text alone does not always contain enough information to unambiguously decide boundaries. For example, the *period* (U+002E FULL STOP) is used ambiguously, sometimes for end-of-sentence purposes, sometimes for abbreviations, and sometimes for numbers. In most cases, however, programmatic text boundaries can match user perceptions quite closely, although sometimes the best that can be done is not to surprise the user.

Rather than concentrate on algorithmically searching for text elements (often called *segments*), a simpler and more useful computation instead detects the *boundaries* (or *breaks*) between those text elements. The determination of those boundaries is often critical to performance, so it is important to be able to make such a determination as quickly as possible. (For a general discussion of text elements, see *Chapter 2, General Structure*, of [Unicode].)

The default boundary determination mechanism specified in this annex provides a straightforward and efficient way to determine some of the most significant boundaries in text: user-perceived characters, words, and sentences. Boundaries used in line breaking (also called *word wrapping*) are defined in [UAX14].

The sheer number of characters in the Unicode Standard, together with its representational power, place requirements on both the specification of text element boundaries and the underlying implementation. The specification needs to allow the designation of large sets of characters sharing the same characteristics (for example, uppercase letters), while the implementation must provide quick access and matches to those large sets. The mechanism also must handle special features of the Unicode Standard, such as nonspacing marks and conjoining jamos.

The default boundary determination builds upon the uniform character representation of the Unicode Standard, while handling the large number of characters and special features such as nonspacing marks and conjoining jamos in an effective manner. As this mechanism lends itself to a completely data-driven implementation, it can be tailored to particular orthographic conventions or user preferences without recoding.

As in other Unicode algorithms, these specifications provide a *logical* description of the processes: implementations can achieve the same results without using code or data that follows these rules step-by-step. In particular, many production-grade implementations will use a state-table approach. In that case, the performance does not depend on the complexity or number of rules. Rather, performance is only affected by the number of characters that may match *after* the boundary position in a rule that applies.

1.1 Notation

A boundary specification summarizes boundary property values used in that specification, then lists the rules for boundary determinations in terms of those property values. The summary is provided as a list, where each element of the list is one of the following:

- A literal character
- A range of literal characters
- All characters satisfying a given condition, using properties defined in the Unicode Character Database [UCD]:

    Non-Boolean property values are given as *<property>* = *<property value>*, such as General_Category = Titlecase_Letter.

    Boolean properties are given as *<property>* = *Yes*, such as Uppercase = Yes.

    Other conditions are specified textually in terms of UCD properties.
- Boolean combinations of the above
- Two special identifiers, *sot* and *eot*, standing for *start of text* and *end of text*, respectively

For example, the following is such a list:

> General_Category = Line_Separator, *or*
> General_Category = Paragraph_Separator, *or*
> General_Category = Control, *or*
> General_Category = Format
> *and not* U+000D CARRIAGE RETURN (CR)
> *and not* U+000A LINE FEED (LF)
> *and not* U+200C ZERO WIDTH NON-JOINER (ZWNJ)
> *and not* U+200D ZERO WIDTH JOINER (ZWJ)

In the table assigning the boundary property values, all of the values are intended to be disjoint except for the special value **Any**. In case of conflict, rows higher in the table have precedence in terms of assigning property values to characters. Data files containing explicit assignments of the property values are found in [Props].

Boundary determination is specified in terms of an ordered list of rules, indicating the status of a boundary position. The rules are numbered for reference and are applied in sequence to determine whether there is a boundary at any given offset. That is, there is an implicit “otherwise” at the front of each rule following the first. The rules are processed from top to bottom. As soon as a rule matches and produces a boundary status (boundary or no boundary) for that offset, the process is terminated.

Each rule consists of a left side, a boundary symbol (see *Table 1*), and a right side. Either of the sides can be empty. The left and right sides use the boundary property values in regular expressions. The regular expression syntax used is a simplified version of the format supplied in *Unicode Technical Standard #18, Unicode Regular Expressions* [UTS18].

Table 1. Boundary Symbols

÷	Boundary (allow break here)
×	No boundary (do not allow break here)
→	Treat whatever on the left side as if it were what is on the right side

An *open-box* symbol (“␣”) is used to indicate a space in examples.

1.2 Rule Constraints

These rules are constrained in three ways, to make implementations significantly simpler and more efficient. These constraints have not been found to be limitations for natural language use. In particular, the rules are formulated so that they can be efficiently implemented, such as with a deterministic finite-state machine based on a small number of property values.

1. ***Single boundaries****.* Each rule has exactly one boundary position. This restriction is more a limitation on the specification methods, because a rule with multiple boundaries could be expressed instead as multiple rules. For example:

   “a b ÷ c d ÷ e f” could be broken into two rules “a b ÷ c d e f” and “a b c d ÷ e f”

   “a b × c d × e f” could be broken into two rules “a b × c d e f” and “a b c d × e f”
2. ***Limited negation****.* Negation of expressions is limited to instances that resolve to a match against single characters, such as “¬(OLetter | Upper | Lower | Sep)”.
3. ***Ignore degenerates****.* No special provisions are made to get marginally better behavior for degenerate cases that never occur in practice, such as an *A* followed by an Indic combining mark.
4. ***Script boundaries****.* Script boundaries are treated as degenerate cases in these rules, so the string “aquaφοβία” is treated as a single word, and the sequence ‘a’ + ‘ ि’ as a single grapheme cluster. However, implementations are free to customize boundary testing to break at script boundaries, which may be especially useful for grapheme clusters. When this is done, the Common/Inherited values need to be handled properly, and the Script_Extensions property should be used instead of the Script property alone.

2 Conformance

There are many different ways to divide text elements corresponding to user-perceived characters, words, and sentences, and the Unicode Standard does not restrict the ways in which implementations can produce these divisions. However, it does provide conformance clauses to enable implementations to clearly describe their behavior in relation to the default behavior.

UAX29-C1. **Extended Grapheme Cluster Boundaries:** *An implementation shall choose either UAX29-C1-1 or UAX29-C1-2 to determine whether an offset within a sequence of characters is an extended grapheme cluster boundary.*

UAX29-C1-1. *Use the property values defined in the Unicode Character Database [UCD] and the* ***extended*** *rules in Section 3.1 Grapheme Cluster Boundary Rules to determine the boundaries.*

The default grapheme clusters are also known as **extended grapheme clusters**.

UAX29-C1-2. *Declare the use of a profile of UAX29-C1-1, and define that profile with a precise specification of any changes in property values or rules and/or provide a description of programmatic overrides to the behavior of UAX29-C1-1.*

Legacy grapheme clusters are such a profile.

UAX29-C2. **Word Boundaries:** *An implementation shall choose either UAX29-C2-1 or UAX29-C2-2 to determine whether an offset within a sequence of characters is a word boundary.*

UAX29-C2-1. *Use the property values defined in the Unicode Character Database [UCD] and the rules in Section 4.1 Default Word Boundary Specification to determine the boundaries.*

UAX29-C2-2. *Declare the use of a profile of UAX29-C2-1, and define that profile with a precise specification of any changes in property values or rules and/or provide a description of programmatic overrides to the behavior of UAX29-C2-1.*

UAX29-C3. **Sentence Boundaries:** *An implementation shall choose either UAX29-C3-1 or UAX29-C3-2 to determine whether an offset within a sequence of characters is a sentence boundary.*

UAX29-C3-1. *Use the property values defined in the Unicode Character Database [UCD] and the rules in Section 5.1 Default Sentence Boundary Specification to determine the boundaries.*

UAX29-C3-2. *Declare the use of a profile of UAX29-C3-1, and define that profile with a precise specification of any changes in property values or rules and/or provide a description of programmatic overrides to the behavior of UAX29-C3-1.*

This specification defines *default* mechanisms; more sophisticated implementations can *and should* tailor them for particular locales or environments and, for the purpose of claiming conformance, document the tailoring in the form of a profile. For example, reliable detection of word boundaries in languages such as Thai, Lao, Chinese, or Japanese requires the use of dictionary lookup or other mechanisms, analogous to English hyphenation. An implementation therefore may need to provide means for a programmatic override of the default mechanisms described in this annex. Note that a profile can both add and remove boundary positions, compared to the results specified by UAX29-C1-1, UAX29-C2-1, or UAX29-C3-1.

Notes:

- Locale-sensitive boundary specifications, including boundary suppressions, can be expressed in LDML [UTS35]. Some profiles are available in the Common Locale Data Repository [CLDR].
- Some changes to rules and data are needed for best segmentation behavior of additional emoji zwj sequences [UTS51]. Implementations are strongly encouraged to use the extended text segmentation rules in the latest version of CLDR.

To maintain canonical equivalence, all of the following specifications are defined on text normalized in form NFD, as defined in Unicode Standard Annex #15, “Unicode Normalization Forms” [UAX15]. Boundaries never occur within a combining character sequence or conjoining sequence, so the boundaries within non-NFD text can be derived from corresponding boundaries in the NFD form of that text. For convenience, the default rules have been written so that they can be applied directly to non-NFD text and yield equivalent results. (This may not be the case with tailored default rules.) For more information, see Section 6, *Implementation Notes*.

3 Grapheme Cluster Boundaries

~~It is important to recognize that what the user thinks of as a “character”—a basic unit of a writing system for a language—may not be just a single Unicode code point. Instead, that basic unit may be made up of multiple Unicode code points. To avoid ambiguity with the computer use of the term *character,* this is called a *user-perceived character*. For example, “G” + *grave-accent* is a *user-perceived character*: users think of it as a single character, yet is actually represented by two Unicode code points. These user-perceived characters are approximated by what is called a *grapheme cluster*, which can be determined programmatically.~~

~~Grapheme cluster boundaries are important for collation, regular expressions, UI interactions, segmentation for vertical text, identification of boundaries for first-letter styling, and counting “character” positions within text. Word boundaries, line boundaries, and sentence boundaries should not occur within a grapheme cluster: in other words, a grapheme cluster should be an atomic unit with respect to the process of determining these other boundaries.~~

~~As far as a user is concerned, the underlying representation of text is not important, but it is important that an editing interface present a uniform implementation of what the user thinks of as characters. Grapheme clusters can be treated as units, by default, for processes such as the formatting of drop caps, as well as the implementation of text selection, arrow key movement or backspacing through text, and so forth. For example, when a grapheme cluster is represented internally by a character sequence consisting of base character + accents, then using the right arrow key would skip from the start of the base character to the end of the last accent.~~

A single Unicode code point is often, but not always the same as a basic unit of a writing system for a language, or what a typical user might think of as a “character”. There are many cases where such a basic unit is made up of multiple Unicode code points. To avoid ambiguity with the term character as defined for encoding purposes, it can be useful to speak of a *user-perceived character*. For example, “G” + grave-accent is a user-perceived character: users think of it as a single character, yet is actually represented by two Unicode code points.

The notion of user-perceived character is not always an unambiguous concept for a given writing system: it may differ based on language, script style, or even based on context, for the same user. Drop-caps and initialisms, text selection, or "character" counting for text size limits are all contexts in which the basic unit may be defined differently.

In implementations, the notion of user-perceived characters corresponds to the concept of grapheme clusters. They are a best-effort approximation that can be determined programmatically and unambiguously. The definition of grapheme clusters attempts to achieve uniformity across all human text without requiring language or font metadata about that text. As an approximation, it may not cover all potential types of user-perceived characters, and it may have suboptimal behavior in some scripts where further metadata is needed, or where a different notion of user-perceived character is preferred. Such special cases may require a customization of the algorithm, while the generic case continues to be supported by the standard algorithm.

As far as a user is concerned, the underlying representation of text is not important, but it is important that an editing interface present a uniform implementation of what the user thinks of as characters. Grapheme clusters can be treated as units, by default, for processes such as the formatting of drop caps, as well as the implementation of text selection, arrow key movement, forward deletion, and so forth. For example, when a grapheme cluster is represented internally by a character sequence consisting of base character + accents, then using the right arrow key would skip from the start of the base character to the end of the last accent.

Grapheme cluster boundaries are also important for collation, regular expressions, UI interactions, segmentation for vertical text, identification of boundaries for first-letter styling, and counting “character” positions within text. Word boundaries, line boundaries, and sentence boundaries should not occur within a grapheme cluster: in other words, a grapheme cluster should be an atomic unit with respect to the process of determining these other boundaries.

This document defines a default specification for grapheme clusters. It may be customized for particular languages, operations, or other situations. For example, arrow key movement could be tailored by language, or could use knowledge specific to particular fonts to move in a more granular manner, in circumstances where it would be useful to edit individual components. This could apply, for example, to the complex editorial requirements for the Northern Thai script Tai Tham (Lanna). Similarly, editing a grapheme cluster element by element may be preferable in some circumstances. For example, on a given system the *backspace key* might delete by code point, while the *delete key* may delete an entire cluster.

Moreover, there is not a one-to-one relationship between grapheme clusters and keys on a keyboard. A single key on a keyboard may correspond to a whole grapheme cluster, a part of a grapheme cluster, or a sequence of more than one grapheme cluster.

Grapheme clusters can only provide an approximation of where to put cursors. Detailed cursor placement depends on the text editing framework. The text editing framework determines where the edges of glyphs are, and how they correspond to the underlying characters, based on information supplied by the lower-level text rendering engine and font. For example, the text editing framework must know if a digraph is represented as a single glyph in the font, and therefore may not be able to position a cursor at the proper position separating its two components. That framework must also be able to determine display representation in cases where two glyphs overlap—this is true generally when a character is displayed together with a subsequent nonspacing mark, but must also be determined in detail for complex script rendering. For cursor placement, grapheme clusters boundaries can only supply an approximate guide for cursor placement using least-common-denominator fonts for the script.

In those relatively rare circumstances where programmers need to supply end users with user-perceived character counts, the counts should correspond to the number of segments delimited by grapheme cluster boundaries. Grapheme clusters *may also be* used in searching and matching; for more information, see Unicode Technical Standard #10, “Unicode Collation Algorithm” [UTS10], and Unicode Technical Standard #18, “Unicode Regular Expressions” [UTS18].

The Unicode Standard provides a default algorithm for determining grapheme cluster boundaries; the default grapheme clusters are also known as **extended grapheme clusters**. For backwards compatibility with earlier versions of this specification, the Standard also defines and maintains a profile for **legacy grapheme clusters**.

These algorithms can be adapted to produce **tailored grapheme clusters** for specific locales or other customizations, such as the contractions used in collation tailoring tables. In *Table 1a* are some examples of the differences between these concepts. The tailored examples are only for illustration: what constitutes a grapheme cluster will depend on the customizations used by the particular tailoring in question.

Table 1a. Sample Grapheme Clusters

Ex	Characters	Comments
Grapheme clusters (both legacy and extended)		
g̈	0067 (g) LATIN SMALL LETTER G 0308 (◌̈) COMBINING DIAERESIS	combining character sequences
각	AC01 (각) HANGUL SYLLABLE GAG	Hangul syllables such as *gag* (which may be a single character, or a sequence of conjoining jamos)
	1100 (ᄀ) HANGUL CHOSEONG KIYEOK 1161 (ᅡ) HANGUL JUNGSEONG A 11A8 (ᆨ) HANGUL JONGSEONG KIYEOK	
ก	0E01 (ก) THAI CHARACTER KO KAI	Thai *ko*
Extended grapheme clusters		
நி	0BA8 (ந) TAMIL LETTER NA 0BBF (◌ி) TAMIL VOWEL SIGN I	Tamil *ni*
เ	0E40 (เ) THAI CHARACTER SARA E	Thai *e*
กำ	0E01 (ก) THAI CHARACTER KO KAI 0E33 (◌ำ) THAI CHARACTER SARA AM	Thai *kam*
षि	0937 (ष) DEVANAGARI LETTER SSA 093F (ि◌) DEVANAGARI VOWEL SIGN I	Devanagari *ssi*
क्षि	0915 (क) DEVANAGARI LETTER KA 094D (◌्) DEVANAGARI SIGN VIRAMA 0937 (ष) DEVANAGARI LETTER SSA 093F (ि◌) DEVANAGARI VOWEL SIGN I	Devanagari *kshi*
Legacy grapheme clusters		

ำ	0E33 (ำ) THAI CHARACTER SARA AM	Thai *am*
ष	0937 (ष) DEVANAGARI LETTER SSA	Devanagari *ssa*
ि	093F (ि) DEVANAGARI VOWEL SIGN I	Devanagari *i*
Possible tailored grapheme clusters in a profile		
ch	0063 (c) LATIN SMALL LETTER C 0068 (h) LATIN SMALL LETTER H	Slovak *ch* digraph
kʷ	006B (k) LATIN SMALL LETTER K 02B7 (ʷ) MODIFIER LETTER SMALL W	sequence with modifier letter

See also: Where is my Character?, and the UCD file ***NamedSequences.txt*** *[Data34].*

A ***legacy grapheme cluster*** is defined as a base (such as A or カ) followed by zero or more continuing characters. One way to think of this is as a sequence of characters that form a “stack”.

The base can be single characters, or be any sequence of Hangul Jamo characters that form a Hangul Syllable, as defined by D133 in The Unicode Standard, or be a pair of Regional_Indicator (RI) characters. For more information about RI characters, see [UTS51].

The continuing characters include nonspacing marks, the Join_Controls (U+200C ZERO WIDTH NON-JOINER and U+200D ZERO WIDTH JOINER) used in Indic languages, and a few spacing combining marks to ensure canonical equivalence. There are cases in Bangla, Khmer, Malayalam, and Odiya in which a ZWNJ occurs after a consonant and before a *virama* or other combining mark. These cases should not provide an opportunity for a grapheme cluster break. Therefore, ZWNJ has been included in the Extend class. Additional cases need to be added for completeness, so that any string of text can be divided up into a sequence of grapheme clusters. Some of these may be *degenerate* cases, such as a control code, or an isolated combining mark.

An ***extended grapheme cluster*** is the same as a legacy grapheme cluster, with the addition of some other characters. The continuing characters are extended to include all spacing combining marks, such as the spacing (but dependent) vowel signs in Indic scripts. For example, this includes U+093F (ि) DEVANAGARI VOWEL SIGN I. The extended grapheme clusters should be used in implementations in preference to legacy grapheme clusters, because they provide better results for Indic scripts such as Tamil or Devanagari in which editing by orthographic syllable is typically preferred. For scripts such as Thai, Lao, and certain other Southeast Asian scripts, editing by visual unit is typically preferred, so for those scripts the behavior of extended grapheme clusters is similar to (but not identical to) the behavior of legacy grapheme clusters.

For the rules defining the boundaries for grapheme clusters, see *Section 3.1*. For more information on the composition of Hangul syllables, see *Chapter 3, Conformance*, of [Unicode].

A key feature of Unicode grapheme clusters (both legacy and extended) is that they remain unchanged across all canonically equivalent forms of the underlying text. Thus the boundaries remain unchanged whether the text is in NFC or NFD. Using a grapheme cluster as the fundamental unit of matching thus provides a very clear and easily explained basis for canonically equivalent matching. This is important for applications from searching to regular expressions.

Another key feature is that default Unicode grapheme clusters are atomic units with respect to the process of determining the Unicode default word, and sentence boundaries. They are usually—but not always—atomic units with respect to line boundaries: there are exceptions due to the special handling of spaces. For more information, see *Section 9.2 Legacy Support for Space Character as Base for Combining Marks* in [UAX14].

Grapheme clusters can be tailored to meet further requirements. Such tailoring is permitted, but the possible rules are outside of the scope of this document. One example of such a tailoring would be for the *aksaras*, or *orthographic syllables*, used in many Indic scripts. Aksaras usually consist of a consonant, sometimes with an inherent vowel and sometimes followed by an explicit, dependent vowel whose rendering may end up on any side of the consonant letter base. Extended grapheme clusters include such simple combinations.

However, aksaras may also include one or more additional consonants, typically with a *virama* (halant) character between each pair of consonants in the sequence. Some consonant cluster aksaras are not incorporated into the default rules for extended grapheme clusters, in part because not all such sequences are considered to be single “characters” by users. Another reason is that additional changes to the rules are made when new information becomes available. Indic scripts vary considerably in how they handle the rendering of such aksaras—in some cases stacking them up into combined forms known as consonant conjuncts, and in other cases stringing them out horizontally, with visible renditions of the halant on each consonant in the sequence. There is even greater variability in how the typical liquid consonants (or “medials”), *ya, ra, la,* and *wa*, are handled for display in combinations in aksaras. So tailorings for aksaras may need to be script-, language-, font-, or context-specific to be useful.

> **Note:** Font-based information may be required to determine the appropriate unit to use for UI purposes, such as identification of boundaries for first-letter paragraph styling. For example, such a unit could be a ligature formed of two grapheme clusters, such as لا (Arabic lam + alef).

The Unicode specification of grapheme clusters >allows for more sophisticated profiles where appropriate. Such definitions may more precisely match the user expectations within individual languages for given processes. For example, “ch” may be considered a grapheme cluster in Slovak, for processes such as collation. The default definitions are, however, designed to provide a much more accurate match to overall user expectations for what the user perceives of as *characters* than is provided by individual Unicode code points.

> **Note:** The term cluster is used to emphasize that the term grapheme is used differently in linguistics.

Display of Grapheme Clusters. Grapheme clusters are not the same as ligatures. For example, the grapheme cluster “ch” in Slovak is not normally a ligature and, conversely, the ligature “fi” is not a grapheme cluster. Default grapheme clusters do not necessarily reflect text display. For example, the sequence <f, i> may be displayed as a single glyph on the screen, but would still be two grapheme clusters.

For information on the matching of grapheme clusters with regular expressions, see Unicode Technical Standard #18, “Unicode Regular Expressions” [UTS18].

Degenerate Cases. The default specifications are designed to be simple to implement, and provide an algorithmic determination of grapheme clusters. However, they do *not* have to cover edge cases that will not occur in practice. For the purpose of segmentation, they may also include degenerate cases that are not thought of as grapheme clusters, such as an isolated control character or combining mark. In this, they differ from the combining character sequences and extended combining character sequences defined in [Unicode]. In addition, Unassigned (Cn) code points and Private_Use (Co) characters are given property values that anticipate potential usage.

Combining Character Sequences and Grapheme Clusters. For comparison, *Table 1b* shows the relationship between combining character sequences and grapheme clusters, using regex notation. Note that given alternates (X|Y), the first match is taken. The simple identifiers starting with lowercase are variables that are defined in *Table 1c*; those starting with uppercase letters are **Grapheme_Cluster_Break Property Values** defined in *Table 2*.

Table 1b. Combining Character Sequences and Grapheme Clusters

Term	Regex	Notes
combining character sequence	`ccs-base? ccs-extend+`	A single base character is not a combining character sequence. However, a single combining mark *is* a (degenerate) combining character sequence.
extended combining character sequence	`extended_base? ccs-extend+`	extended_base includes Hangul Syllables

legacy grapheme cluster	`crlf`
	`| Control`
	`| legacy-core legacy-postcore*`
		A single base character is a grapheme cluster. Degenerate cases include any isolated non-base characters, and non-base characters like controls.

extended grapheme cluster	`crlf`
	`| Control`
	`| precore* core postcore*`
		Extended grapheme clusters add prepending and spacing marks.

Table 1b uses several symbols defined in *Table 1c*. Square brackets and \p{...} are used to indicate sets of characters, using the normal UnicodeSet notion.

Table 1c. Regex Definitions

`ccs-base :=`	`[\p{L}\p{N}\p{P}\p{S}\p{Zs}]`
`ccs-extend :=`	`[\p{M}\p{Join_Control}]`

`extended_base :=`	`ccs-base`
	`| hangul-syllable`

`crlf :=`	`CR LF | CR | LF`

`legacy-core :=`	`hangul-syllable`
	`| RI-Sequence`
	`| xpicto-sequence`
	`| [^Control CR LF]`

`legacy-postcore :=`	`[Extend ZWJ]`

`core :=`	`hangul-syllable`
	`| RI-Sequence`
	`| xpicto-sequence`
	`| conjunctCluster`
	`| [^Control CR LF]`

`postcore :=`	`[Extend ZWJ SpacingMark]`
`precore :=`	`Prepend`
`RI-Sequence :=`	`RI RI`

`hangul-syllable :=`	`L* (V+ | LV V* | LVT) T*`
	`| L+`
	`| T+`

`xpicto-sequence :=`	`\p{Extended_Pictographic} (Extend* ZWJ \p{Extended_Pictographic})*`
`conjunctCluster :=`	`\p{InCB=Consonant} ([\p{InCB=Extend} \p{InCB=Linker}]* \p{InCB=Linker} [\p{InCB=Extend} \p{InCB=Linker}]* \p{InCB=Consonant})+`

3.1 Default Grapheme Cluster Boundary Specification

The following is a general specification for grapheme cluster boundaries—language-specific rules in [CLDR] should be used where available.

The Grapheme_Cluster_Break property value assignments are explicitly listed in the corresponding data file in [Props]. The values in that file are the normative property values.

For illustration, property values are summarized in *Table 2*, but the lists of characters are illustrative.

Table 2. Grapheme_Cluster_Break Property Values

Value	Summary List of Characters
CR	U+000D CARRIAGE RETURN (CR)
LF	U+000A LINE FEED (LF)

Control	General_Category = Line_Separator, *or*
	General_Category = Paragraph_Separator, *or*
	General_Category = Control, *or*
	General_Category = Unassigned *and* Default_Ignorable_Code_Point, *or*
	General_Category = Format
	and not U+000D CARRIAGE RETURN
	and not U+000A LINE FEED
	and not U+200C ZERO WIDTH NON-JOINER (ZWNJ)
	and not U+200D ZERO WIDTH JOINER (ZWJ)
	and not Prepended_Concatenation_Mark = Yes

Extend	Grapheme_Extend = Yes, *or* *Emoji_Modifier=Yes* *This includes:* General_Category = Nonspacing_Mark General_Category = Enclosing_Mark U+200C ZERO WIDTH NON-JOINER *plus a few* General_Category = Spacing_Mark *needed for canonical equivalence.*
ZWJ	U+200D ZERO WIDTH JOINER
Regional_Indicator (RI)	Regional_Indicator = Yes

	This consists of the range: U+1F1E6 REGIONAL INDICATOR SYMBOL LETTER A ..U+1F1FF REGIONAL INDICATOR SYMBOL LETTER Z

Prepend	Indic_Syllabic_Category = Consonant_Preceding_Repha*, or*
	Indic_Syllabic_Category = Consonant_Prefixed*, or*
	Prepended_Concatenation_Mark = Yes

SpacingMark	Grapheme_Cluster_Break ≠ Extend*, and* General_Category = Spacing_Mark*, or* *any of the following (which have* General_Category = Other_Letter*):* U+0E33 (ำ) THAI CHARACTER SARA AM U+0EB3 (ຳ) LAO VOWEL SIGN AM *Exceptions: The following (which have* General_Category = Spacing_Mark *and would otherwise be included) are specifically excluded:* U+102B (◌ါ) MYANMAR VOWEL SIGN TALL AA U+102C (◌ာ) MYANMAR VOWEL SIGN AA U+1038 (◌း) MYANMAR SIGN VISARGA U+1062 (◌ၢ) MYANMAR VOWEL SIGN SGAW KAREN EU ..U+1064 (◌ၤ) MYANMAR TONE MARK SGAW KAREN KE PHO U+1067 (◌ၧ) MYANMAR VOWEL SIGN WESTERN PWO KAREN EU ..U+106D (◌ၭ) MYANMAR SIGN WESTERN PWO KAREN TONE-5 U+1083 (◌ႃ) MYANMAR VOWEL SIGN SHAN AA U+1087 (◌ႇ) MYANMAR SIGN SHAN TONE-2 ..U+108C (◌ႌ) MYANMAR SIGN SHAN COUNCIL TONE-3 U+108F (◌ႏ) MYANMAR SIGN RUMAI PALAUNG TONE-5 U+109A (◌ႚ) MYANMAR SIGN KHAMTI TONE-1 ..U+109C (◌ႜ) MYANMAR VOWEL SIGN AITON A U+1A61 (□) TAI THAM VOWEL SIGN A U+1A63 (□) TAI THAM VOWEL SIGN AA U+1A64 (□) TAI THAM VOWEL SIGN TALL AA U+AA7B (◌ꩻ) MYANMAR SIGN PAO KAREN TONE U+AA7D (◌ꩽ) MYANMAR SIGN TAI LAING TONE-5 U+11720 (□) AHOM VOWEL SIGN A U+11721 (□) AHOM VOWEL SIGN AA
L	Hangul_Syllable_Type=L*, such as:* U+1100 (ᄀ) HANGUL CHOSEONG KIYEOK U+115F ( ) HANGUL CHOSEONG FILLER U+A960 (ꥠ) HANGUL CHOSEONG TIKEUT-MIEUM U+A97C (ꥼ) HANGUL CHOSEONG SSANGYEORINHIEUH

V	Hangul_Syllable_Type=V, *such as:* U+1160 (ᅠ) HANGUL JUNGSEONG FILLER U+11A2 (ᆢ) HANGUL JUNGSEONG SSANGARAEA U+D7B0 (ힰ) HANGUL JUNGSEONG O-YEO U+D7C6 (ퟆ) HANGUL JUNGSEONG ARAEA-E*, and:* U+16D63 (□) KIRAT RAI VOWEL SIGN AA U+16D67 (□) KIRAT RAI VOWEL SIGN E ..U+16D6A (□) KIRAT RAI VOWEL SIGN AU
T	Hangul_Syllable_Type=T, *such as:* U+11A8 (ᆨ) HANGUL JONGSEONG KIYEOK U+11F9 (ᇹ) HANGUL JONGSEONG YEORINHIEUH U+D7CB (ퟋ) HANGUL JONGSEONG NIEUN-RIEUL U+D7FB (ퟻ) HANGUL JONGSEONG PHIEUPH-THIEUTH
LV	Hangul_Syllable_Type=LV, *that is:* U+AC00 (가) HANGUL SYLLABLE GA U+AC1C (개) HANGUL SYLLABLE GAE U+AC38 (갸) HANGUL SYLLABLE GYA ...
LVT	Hangul_Syllable_Type=LVT, *that is:* U+AC01 (각) HANGUL SYLLABLE GAG U+AC02 (갂) HANGUL SYLLABLE GAGG U+AC03 (갃) HANGUL SYLLABLE GAGS U+AC04 (간) HANGUL SYLLABLE GAN ...
E_Base	*This value is obsolete and unused.*
E_Modifier	*This value is obsolete and unused.*
Glue_After_Zwj	*This value is obsolete and unused.*
E_Base_GAZ (EBG)	*This value is obsolete and unused.*
Any	*This is not a property value; it is used in the rules to represent any code point.*

3.1.1 Grapheme Cluster Boundary Rules

The same rules are used for the two variants of grapheme clusters, except the rules GB9a, GB9b, and GB9c. The following table shows the differences, which are also marked on the rules themselves. The extended rules are recommended, except where the legacy variant is required for a specific environment.

Grapheme Cluster Variant	Includes	Excludes
LG: legacy grapheme clusters		GB9a, GB9b, GB9c
EG: extended grapheme clusters	GB9a, GB9b, GB9c	

When citing the Unicode definition of grapheme clusters, it must be clear which of the two alternatives are being specified: extended versus legacy.

Break at the start and end of text, unless the text is empty.			
GB1	sot	÷	Any
GB2	Any	÷	eot
Do not break between a CR and LF. Otherwise, break before and after controls.			
GB3	CR	×	LF
GB4	(Control | CR | LF)	÷	
GB5		÷	(Control | CR | LF)
Do not break Hangul syllable or other conjoining sequences.			
GB6	L	×	(L | V | LV | LVT)
GB7	(LV | V)	×	(V | T)
GB8	(LVT | T)	×	T
Do not break before extending characters or ZWJ.			
GB9		×	(Extend | ZWJ)
The GB9a and GB9b rules only apply to extended grapheme clusters: *Do not break before SpacingMarks, or after Prepend characters.*			
GB9a		×	SpacingMark
GB9b	Prepend	×	
The GB9c rule only applies to extended grapheme clusters: *Do not break within certain combinations with Indic_Conjunct_Break (InCB)=Linker.*			
GB9c	\p{InCB=Consonant} [\p{InCB=Extend} \p{InCB=Linker}]* \p{InCB=Linker} [\p{InCB=Extend} \p{InCB=Linker}]*	×	\p{InCB=Consonant}
Do not break within emoji modifier sequences or emoji zwj sequences.			
GB11	\p{Extended_Pictographic} Extend* ZWJ	×	\p{Extended_Pictographic}
Do not break within emoji flag sequences. That is, do not break between regional indicator (RI) symbols if there is an odd number of RI characters before the break point.			
GB12	sot (RI RI)* RI	×	RI
GB13	[^RI] (RI RI)* RI	×	RI
Otherwise, break everywhere.			
GB999	Any	÷	Any

Notes:

- Grapheme cluster boundaries can be transformed into simple regular expressions. For more information, see *Section 6.3, State Machines* and *Table 1c, Regex Definitions*.

- The Grapheme_Base and Grapheme_Extend properties predated the development of the Grapheme_Cluster_Break property. The set of characters with Grapheme_Extend=Yes is used to derive the set of characters with Grapheme_Cluster_Break=Extend. However, the Grapheme_Base property proved to be insufficient for determining grapheme cluster boundaries. Grapheme_Base is no longer used by this specification.
- Each *emoji sequence* is a single grapheme cluster. See definition ED-17 in Unicode Technical Standard #51, "Unicode Emoji" [UAX51].
- Similar to Jamo clustering into Hangul Syllables, other characters bind tightly into grapheme clusters, that, unlike combining characters, don't depend on a base character. These characters are said to exhibit *conjoining behavior*. For the purpose of Grapheme_Cluster_Break, the property value V has been extended beyond characters of Hangul_Syllable_Type=V to cover them.

4 Word Boundaries

Word boundaries are used in a number of different contexts. The most familiar ones are selection (double-click mouse selection), cursor movement (“move to next word” control-arrow keys), and the dialog option “Whole Word Search” for search and replace. They are also used in database queries, to determine whether elements are within a certain number of words of one another. Searching may also use word boundaries in determining matching items. Word boundaries are not restricted to whitespace and punctuation. Indeed, some languages do not use spaces at all.

Figure 1 gives an example of word boundaries, marked in the sample text with vertical bars. In the following discussion, search terms are indicated by enclosing them in square brackets for clarity. Spaces are indicated with the open-box symbol “␣”, and the matching parts between the search terms and target text are emphasized in color.

Figure 1. Word Boundaries

The		quick		(	“	brown	”	)		fox		can’t		jump		32.3		feet	,		right	?

Boundaries such as those flanking the words in *Figure 1* are the boundaries that users would expect, for example, when searching for a term in the target text using Whole Word Search mode. In that mode there is a match if—in addition to a matching sequence of characters—there are word boundaries in the target text on both sides of the search term. In the sample target text in *Figure 1*, Whole Word Search would have results such as the following:

- The search term [brown] matches because there are word boundaries on both sides.
- The search term [brow] does not match because there is no word boundary in the target text between ‘w’ and the following character, ‘n’.
- The term [“brown”] matches because there are word boundaries between the quotation marks and the parentheses that enclose them.
- The term [(“brown”)] also matches because there are word boundaries between the parentheses and the space characters around them.
- Finally, the term [␣(“brown”)␣] with spaces included matches as well, because there are word boundaries between the space characters and the letters immediately before and after them in the target text.

To allow for such matches that users would expect, there are word breaks by default between most characters that are not normally considered parts of words, such as punctuation and spaces.

Word boundaries can also be used in intelligent cut and paste. With this feature, if the user cuts a selection of text on word boundaries, adjacent spaces are collapsed to a single space. For example, cutting “quick” from “The␣quick␣fox” would leave “The␣␣fox”. Intelligent cut and paste collapses this text to “The␣fox”. However, spaces need to be handled separately: cutting the center space from “The␣␣␣fox” probably should not collapse the remaining two spaces to one.

Proximity tests in searching determines whether, for example, “quick” is within three words of “fox”. That is done with the above boundaries by ignoring any words that contain only whitespace, punctuation, and similar characters, as in *Figure 2*. Thus, for proximity, “fox” is within three words of “quick”. This same technique can be used for “get next/previous word” commands or keyboard arrow keys. Letters are not the only characters that can be used to determine the “significant” words; different implementations may include other types of characters such as digits or perform other analysis of the characters.

Figure 2. Extracted Words

The	quick	brown	fox	can’t	jump	32.3	feet	right

~~Word boundaries are related to line boundaries, but are distinct: there are some word boundaries that are not line boundaries, and vice versa. A line boundary is usually a word boundary, but there are exceptions such as a word containing a SHY (soft hyphen): it will break across lines, yet is a single word.~~

As with the other default specifications, implementations may override (tailor) the results to meet the requirements of different environments or particular languages. For some languages, it may also be necessary to have different tailored word break rules for selection versus Whole Word Search.

~~In particular, the characters with the Line_Break property values of Contingent_Break (CB), Complex_Context (SA/Southeast Asian), and Unknown (XX) are assigned Word_Break property values based on criteria outside of the scope of this annex. The relationship of line break and word break boundaries is script-specific and may require special handling for satisfactory treatment.~~

Whether the default word boundary detection described here is adequate, and whether word boundaries are related to line breaks, varies between scripts. The style of context analysis in line breaking (see [UAX14, section 3.1]) used for a script can provide some rough guidance:

- For scripts that use the Western style of context analysis, default word boundaries and default line breaks are usually adequate. A default line boundary break opportunity is usually a default word boundary, but there are exceptions such as a word containing a SHY (soft hyphen): it will break across lines, yet is a single word. Tailorings may find additional line break opportunities within words due to hyphenation. Scripts in this group include Latin, Arabic, Devanagari, and many others; they can be identified by having letters with line break class AL.
- For scripts that use the East Asian or Brahmic styles of context analysis, the default word boundary detection is not adequate; it needs tailoring. The default line breaks, on the other hand, are usually adequate. Word boundaries are irrelevant to line breaking. Scripts in this group include Chinese, Japanese, Brahmi, Javanese, and others; they can be identified by having letters with line break class ID, AK, or AS.
- For scripts that use the South East Asian style of context analysis, neither the default word boundaries nor the default line breaks are adequate. Both need tailoring. The reason is that line breaks should only occur at word boundaries, but there’s no

demarcation of words. Scripts in this group include Thai, Myanmar, Khmer, and others; they can be identified by having letters with line break class SA.

Hangul is treated as part of the first group for default word boundary detection; and as part of the second group for default line breaking. Some scripts may be treated as being part of the first group only because not enough information is available for them.

4.1 Default Word Boundary Specification

The following is a general specification for word boundaries—language-specific rules in [CLDR] should be used where available.

The Word_Break property value assignments are explicitly listed in the corresponding data file in [Props]. The values in that file are the normative property values.

For illustration, property values are summarized in *Table 3*, but the lists of characters are illustrative.

Table 3. Word_Break Property Values

Value	Summary List of Characters
CR	U+000D CARRIAGE RETURN (CR)
LF	U+000A LINE FEED (LF)
Newline	U+000B LINE TABULATION U+000C FORM FEED (FF) U+0085 NEXT LINE (NEL) U+2028 LINE SEPARATOR U+2029 PARAGRAPH SEPARATOR

Extend	Grapheme_Extend = Yes, *or*
	General_Category = Spacing_Mark, *or*
	Emoji_Modifier=Yes
	and not U+200D ZERO WIDTH JOINER (ZWJ)

ZWJ	U+200D ZERO WIDTH JOINER
Regional_Indicator (RI)	Regional_Indicator = Yes *This consists of the range:* U+1F1E6 REGIONAL INDICATOR SYMBOL LETTER A ..U+1F1FF REGIONAL INDICATOR SYMBOL LETTER Z
Format	General_Category = Format *and not* U+200B ZERO WIDTH SPACE (ZWSP) *and not* U+200C ZERO WIDTH NON-JOINER (ZWNJ) *and not* U+200D ZERO WIDTH JOINER (ZWJ) *and not* Grapheme_Cluster_Break = Prepend
Katakana	Script = KATAKANA, *or* *any of the following:* U+3031 (〱) VERTICAL KANA REPEAT MARK U+3032 (〲) VERTICAL KANA REPEAT WITH VOICED SOUND MARK U+3033 (〳) VERTICAL KANA REPEAT MARK UPPER HALF U+3034 (〴) VERTICAL KANA REPEAT WITH VOICED SOUND

	MARK UPPER HALF U+3035 (〵) VERTICAL KANA REPEAT MARK LOWER HALF U+309B (゛) KATAKANA-HIRAGANA VOICED SOUND MARK U+309C (゜) KATAKANA-HIRAGANA SEMI-VOICED SOUND MARK U+30A0 (゠) KATAKANA-HIRAGANA DOUBLE HYPHEN U+30FC (ー) KATAKANA-HIRAGANA PROLONGED SOUND MARK U+FF70 (ｰ) HALFWIDTH KATAKANA-HIRAGANA PROLONGED SOUND MARK
Hebrew_Letter	Script = Hebrew *and* General_Category = Other_Letter
ALetter	Alphabetic = Yes, *or* *any of the following characters:* U+02C2 (˂) MODIFIER LETTER LEFT ARROWHEAD ..U+02C5 (˅) MODIFIER LETTER DOWN ARROWHEAD U+02D2 (˒) MODIFIER LETTER CENTRED RIGHT HALF RING ..U+02D7 (˗) MODIFIER LETTER MINUS SIGN U+02DE (˞) MODIFIER LETTER RHOTIC HOOK U+02DF (˟) MODIFIER LETTER CROSS ACCENT U+02E5 (˥) MODIFIER LETTER EXTRA-HIGH TONE BAR ..U+02EB (˫) MODIFIER LETTER YANG DEPARTING TONE MARK U+02ED (˭) MODIFIER LETTER UNASPIRATED U+02EF (˯) MODIFIER LETTER LOW DOWN ARROWHEAD ..U+02FF (˿) MODIFIER LETTER LOW LEFT ARROW U+055A (՚) ARMENIAN APOSTROPHE U+055B (՛) ARMENIAN EMPHASIS MARK U+055C (՜) ARMENIAN EXCLAMATION MARK U+055E (՞) ARMENIAN QUESTION MARK U+058A (֊) ARMENIAN HYPHEN U+05F3 (׳) HEBREW PUNCTUATION GERESH U+070F (܏) SYRIAC ABBREVIATION MARK U+A708 (꜈) MODIFIER LETTER EXTRA-HIGH DOTTED TONE BAR ..U+A716 (꜖) MODIFIER LETTER EXTRA-LOW LEFT-STEM TONE BAR U+A720 (꜠) MODIFIER LETTER STRESS AND HIGH TONE U+A721 (꜡) MODIFIER LETTER STRESS AND LOW TONE U+A789 (꞉) MODIFIER LETTER COLON U+A78A (꞊) MODIFIER LETTER SHORT EQUALS SIGN U+AB5B (꭛) MODIFIER BREVE WITH INVERTED BREVE *and* Ideographic = No *and* Word_Break ≠ Katakana *and* Line_Break ≠ Complex_Context (SA) *and* Script ≠ Hiragana *and* Word_Break ≠ Extend *and* Word_Break ≠ Hebrew_Letter
Single_Quote	U+0027 (') APOSTROPHE
Double_Quote	U+0022 (") QUOTATION MARK

MidNumLet	U+002E (.) FULL STOP U+2018 (‘) LEFT SINGLE QUOTATION MARK U+2019 (’) RIGHT SINGLE QUOTATION MARK U+2024 (․) ONE DOT LEADER U+FE52 (﹒) SMALL FULL STOP U+FF07 (＇) FULLWIDTH APOSTROPHE U+FF0E (．) FULLWIDTH FULL STOP
MidLetter	U+003A (:) COLON *(used in Swedish)* U+00B7 (·) MIDDLE DOT U+0387 (·) GREEK ANO TELEIA U+055F (՟) ARMENIAN ABBREVIATION MARK U+05F4 (״) HEBREW PUNCTUATION GERSHAYIM U+2027 (‧) HYPHENATION POINT U+FE13 (︓) PRESENTATION FORM FOR VERTICAL COLON U+FE55 (﹕) SMALL COLON U+FF1A (：) FULLWIDTH COLON
MidNum	Line_Break = Infix_Numeric, *or* *any of the following:* U+066C (٬) ARABIC THOUSANDS SEPARATOR U+FE50 (﹐) SMALL COMMA U+FE54 (﹔) SMALL SEMICOLON U+FF0C (，) FULLWIDTH COMMA U+FF1B (；) FULLWIDTH SEMICOLON *and not* U+003A (:) COLON *and not* U+FE13 (︓) PRESENTATION FORM FOR VERTICAL COLON *and not* U+002E (.) FULL STOP
Numeric	Line_Break = Numeric *or* General_Category = Decimal_Number *and not* U+066C (٬) ARABIC THOUSANDS SEPARATOR
ExtendNumLet	General_Category = Connector_Punctuation, *or* U+202F NARROW NO-BREAK SPACE (NNBSP)
E_Base	*This value is obsolete and unused.*
E_Modifier	*This value is obsolete and unused.*
Glue_After_Zwj	*This value is obsolete and unused.*
E_Base_GAZ (EBG)	*This value is obsolete and unused.*
WSegSpace	General_Category = Zs *and not* Linebreak = Glue
Any	*This is not a property value; it is used in the rules to represent any code point.*

4.1.1 Word Boundary Rules

The table of word boundary rules uses the macro values listed in Table 3a. Each macro represents a repeated union of the basic Word_Break property values and is shown in boldface to distinguish it from the basic property values.

Table 3a. Word_Break Rule Macros

Macro	Represents
AHLetter	(ALetter | Hebrew_Letter)
MidNumLetQ	(MidNumLet | Single_Quote)

Break at the start and end of text, unless the text is empty.			
WB1	sot	÷	Any
WB2	Any	÷	eot
Do not break within CRLF.			
WB3	CR	×	LF
Otherwise break before and after Newlines (including CR and LF)			
WB3a	(Newline | CR | LF)	÷	
WB3b		÷	(Newline | CR | LF)
Do not break within emoji zwj sequences.			
WB3c	ZWJ	×	\p{Extended_Pictographic}
Keep horizontal whitespace together.			
WB3d	WSegSpace	×	WSegSpace
Ignore Format and Extend characters, except after sot, CR, LF, and Newline. (See Section 6.2, Replacing Ignore Rules.) This also has the effect of: Any × (Format | Extend | ZWJ)			
WB4	X (Extend | Format | ZWJ)*	→	X
Do not break between most letters.			
WB5	**AHLetter**	×	**AHLetter**
Do not break letters across certain punctuation, such as within “e.g.” or “example.com”.			
WB6	**AHLetter**	×	(MidLetter | **MidNumLetQ**) **AHLetter**
WB7	**AHLetter** (MidLetter | **MidNumLetQ**)	×	**AHLetter**
WB7a	Hebrew_Letter	×	Single_Quote
WB7b	Hebrew_Letter	×	Double_Quote Hebrew_Letter
WB7c	Hebrew_Letter Double_Quote	×	Hebrew_Letter
Do not break within sequences of digits, or digits adjacent to letters (“3a”, or “A3”).			
WB8	Numeric	×	Numeric
WB9	**AHLetter**	×	Numeric
WB10	Numeric	×	**AHLetter**

Do not break within sequences, such as “3.2” or “3,456.789”.			
WB11	Numeric (MidNum | **MidNumLetQ**)	×	Numeric
WB12	Numeric	×	(MidNum | **MidNumLetQ**) Numeric
Do not break between Katakana.			
WB13	Katakana	×	Katakana
Do not break from extenders.			
WB13a	(**AHLetter** | Numeric | Katakana | ExtendNumLet)	×	ExtendNumLet
WB13b	ExtendNumLet	×	(**AHLetter** | Numeric | Katakana)
Do not break within emoji flag sequences. That is, do not break between regional indicator (RI) symbols if there is an odd number of RI characters before the break point.			
WB15	sot (RI RI)* RI	×	RI
WB16	[^RI] (RI RI)* RI	×	RI
Otherwise, break everywhere (including around ideographs).			
WB999	Any	÷	Any

Notes:

- It is not possible to provide a uniform set of rules that resolves all issues across languages or that handles all ambiguous situations within a given language. The goal for the specification presented in this annex is to provide a workable default; tailored implementations can be more sophisticated.

- ~~For Thai, Lao, Khmer, Myanmar, and other scripts that do not typically use spaces between words, a good implementation should not depend on the default word boundary specification. It should use a more sophisticated mechanism, as is also required for line breaking. Ideographic scripts such as Japanese and Chinese are even more complex. Where Hangul text is written without spaces, the same applies. However, in the absence of a more sophisticated mechanism, the rules specified in this annex supply a well-defined default.~~

- The correct interpretation of hyphens in the context of word boundaries is challenging. It is quite common for separate words to be connected with a hyphen: “out-of-the-box,” “under-the-table,” “Italian-American,” and so on. A significant number are hyphenated names, such as “Smith-Hawkins.” When doing a Whole Word Search or query, users expect to find the word within those hyphens. While there are some cases where they are separate words (usually to resolve some ambiguity such as “re-sort” as opposed to “resort”), it is better overall to keep the hyphen out of the default definition. Hyphens include U+002D HYPHEN-MINUS, U+2010 HYPHEN, possibly also U+058A ARMENIAN HYPHEN, and U+30A0 KATAKANA-HIRAGANA DOUBLE HYPHEN.

- Implementations may build on the information supplied by word boundaries. For example, a spell-checker would first check that each word was valid according to the above definition, checking the four words in “out-of-the-box.” If any of the words failed, it could build the compound word and check if it as a whole sequence was in the dictionary (even if all the components were not in the dictionary), such as with “re-iterate.” Of course, spell-checkers for highly inflected or agglutinative languages will need much more sophisticated algorithms.

- The use of the apostrophe is ambiguous. It is usually considered part of one word (“can’t” or “aujourd’hui”) but it may also be considered as part of two words (“l’objectif”). A further complication is the use of the same character as an apostrophe and as a quotation mark. Therefore leading or trailing apostrophes are best excluded from the default definition of a word. In some languages, such as French and Italian, tailoring to break words when the character after the apostrophe is a vowel may yield better results in more cases. This can be done by adding a rule WB5a.

  Break between apostrophe and vowels (French, Italian).

WB5a	*apostrophe*	÷	vowels

  and defining appropriate property values for apostrophe and vowels. Apostrophe includes U+0027 (') APOSTROPHE and U+2019 (’) RIGHT SINGLE QUOTATION MARK (curly apostrophe). Finally, in some transliteration schemes, apostrophe is used at the beginning of words, requiring special tailoring.

- Certain cases such as colons in words (for example, “AIK:are” and “c:a”) are included in the default even though they may be specific to relatively small user communities (Swedish) because they do not occur otherwise, in normal text, and so do not cause a problem for other languages.

- For Hebrew, a tailoring may include a double quotation mark between letters, because legacy data may contain that in place of U+05F4 (״) HEBREW PUNCTUATION GERSHAYIM. This can be done by adding double quotation mark to MidLetter. U+05F3 (׳) HEBREW PUNCTUATION GERESH may also be included in a tailoring.

- Format characters are included if they are not initial. Thus <LRM><ALetter> will break before the <letter>, but there is no break in <ALetter><LRM><ALetter> or <ALetter><LRM>.

- Characters such as hyphens, apostrophes, quotation marks, and colon should be taken into account when using identifiers that are intended to represent words of one or more natural languages. See Section 2.4, *Specific Character Adjustments*, of [UAX31]. Treatment of hyphens, in particular, may be different in the case of processing identifiers than when using word break analysis for a Whole Word Search or query, because when handling identifiers the goal will be to parse maximal units corresponding to natural language “words,” rather than to find smaller word units within longer lexical units connected by hyphens.

- Normally word breaking does not require breaking between different scripts. However, adding that capability may be useful in combination with other extensions of word segmentation. For example, in Korean the sentence “I live in Chicago.” is written as three segments delimited by spaces:

  - 나는 Chicago에 산다.

  According to Korean standards, the grammatical suffixes, such as “에” meaning “in”, are considered separate words. Thus the above sentence would be broken into the following five words:

  - 나, 는, Chicago, 에, and 산다.

Separating the first two words requires a dictionary lookup, but for Latin text (“Chicago”) the separation is trivial based on the script boundary.

- Modifier letters (General_Category = Lm) are almost all included in the ALetter class, by virtue of their Alphabetic property value. Thus, by default, modifier letters do not cause word breaks and should be included in word selections. Modifier symbols (General_Category = Sk) are not in the ALetter class and so do cause word breaks by default.

- Some or all of the following characters may be tailored to be in MidLetter, depending on the environment:
  - U+002D (-) HYPHEN-MINUS
    U+055A (՚) ARMENIAN APOSTROPHE
    U+058A (֊) ARMENIAN HYPHEN
    U+0F0B (་) TIBETAN MARK INTERSYLLABIC TSHEG
    U+1806 (᠆) MONGOLIAN TODO SOFT HYPHEN
    U+2010 (‐) HYPHEN
    U+2011 (‑) NON-BREAKING HYPHEN
    U+201B (‛) SINGLE HIGH-REVERSED-9 QUOTATION MARK
    U+30A0 (゠) KATAKANA-HIRAGANA DOUBLE HYPHEN
    U+30FB (・) KATAKANA MIDDLE DOT
    U+FE63 (﹣) SMALL HYPHEN-MINUS
    U+FF0D (－) FULLWIDTH HYPHEN-MINUS
  - In UnicodeSet notation, this is: [\u002D\uFF0D\uFE63\u058A\u1806\u2010\u2011\u30A0\u30FB\u201B\u055A\u0F0B]
  - For example, some writing systems use a hyphen character between syllables within a word. An example is the Iu Mien language written with the Thai script. Such words should behave as single words for the purpose of selection (“double-click”), indexing, and so forth, meaning that they should not word-break on the hyphen.
- Some or all of the following characters may be tailored to be in MidNum, depending on the environment, to allow for languages that use spaces as thousands separators, such as €1 234,56.
  - U+0020 SPACE
    U+00A0 NO-BREAK SPACE
    U+2007 FIGURE SPACE
    U+2008 PUNCTUATION SPACE
    U+2009 THIN SPACE
    U+202F NARROW NO-BREAK SPACE
  - In UnicodeSet notation, this is: [\u0020\u00A0\u2007\u2008\u2009\u202F]

4.2 Name Validation

Related to word determination is the issue of *personal name validation*. Implementations sometimes need to validate fields in which personal names are entered. The goal is to distinguish between characters like those in “James Smith-Faley, Jr.” and those in “!#@♥≠”. It is important to be reasonably lenient, because users need to be able to add legitimate names, like “di Silva”, even if the names contain characters such as *space*. Typically, these personal name validations should not be language-specific; someone might be using a Web site in one language while his name is in a different language, for example. A basic set of name validation characters consists the characters allowed in words according to the above definition, plus a number of exceptional characters:

Basic Name Validation Characters

- [\p{name=/COMMA/}\p{name=/FULL STOP/}&\p{p}
  \p{whitespace}-\p{c}
  \p{alpha}
  \p{wb=Katakana}\p{wb=Extend}\p{wb=ALetter}\p{wb=MidLetter}\p{wb=MidNumLet}
  [\u002D\u055A\u058A\u0F0B\u1806\u2010\u2011\u201B\u2E17\u30A0\u30FB\uFE63\uFF0D]
  ]

This is only a basic set of validation characters; in particular, the following points should be kept in mind:

- It is a lenient, non-language-specific set, and could be tailored where only a limited set of languages are permitted, or for other environments. For example, the set can be narrowed if name fields are separated: “,” and “.” may not be necessary if titles are not allowed.
- It includes characters that may not be appropriate for identifiers, and some that would not be parts of words. It also permits some characters that may be part of words in a broad sense, but not part of names, such as in “AIK:are” and “c:a” in Swedish, or hyphenation points used in dictionary words.
- Additional tests may be needed in cases where security is at issue. In particular, names may be validated by transforming them to NFC format, and then testing to ensure that no characters in the result of the transformation change under NFKC. A second test is to use the information in *Table 5. Recommended Scripts* in *Unicode Identifier and Pattern Syntax* [UAX31]. If the name has one or more characters with explicit script values that are not in *Table 5*, then reject the name.

5 Sentence Boundaries

Sentence boundaries are often used for triple-click or some other method of selecting or iterating through blocks of text that are larger than single words. They are also used to determine whether words occur within the same sentence in database queries.

Plain text provides inadequate information for determining good sentence boundaries. Periods can signal the end of a sentence, indicate abbreviations, or be used for decimal points, for example. Without much more sophisticated analysis, one cannot distinguish between the two following examples of the sequence <?, ”, space, uppercase-letter>. In the first example, they mark the end of a sentence, while in the second they do not.

He said, “Are you going?” John shook his head.

“Are you going?” John asked.

Without analyzing the text semantically, it is impossible to be certain which of these usages is intended (and sometimes ambiguities still remain). However, in most cases a straightforward mechanism works well.

> **Note:** As with the other default specifications, implementations are free to override (tailor) the results to meet the requirements of different environments or particular languages. For example, locale-sensitive boundary suppression specifications can be expressed in LDML [UTS35]. Specific sentence boundary suppressions are available in

the Common Locale Data Repository [CLDR] and may be used to improve the quality of boundary analysis.

5.1 Default Sentence Boundary Specification

The following is a general specification for sentence boundaries—language-specific rules in [CLDR] should be used where available.

The Sentence_Break property value assignments are explicitly listed in the corresponding data file in [Props]. The values in that file are the normative property values.

For illustration, property values are summarized in *Table 4*, but the lists of characters are illustrative.

Table 4. Sentence_Break Property Values

Value	Summary List of Characters
CR	U+000D CARRIAGE RETURN (CR)
LF	U+000A LINE FEED (LF)

Extend	Grapheme_Extend = Yes, *or*
	U+200D ZERO WIDTH JOINER (ZWJ), *or*
	General_Category = Spacing_Mark

Sep	U+0085 NEXT LINE (NEL) U+2028 LINE SEPARATOR U+2029 PARAGRAPH SEPARATOR
Format	General_Category = Format *and not* U+200C ZERO WIDTH NON-JOINER (ZWNJ) *and not* U+200D ZERO WIDTH JOINER (ZWJ)
Sp	White_Space = Yes *and* Sentence_Break ≠ Sep *and* Sentence_Break ≠ CR *and* Sentence_Break ≠ LF
Lower	Lowercase = Yes *and* Grapheme_Extend = No *and* not in the ranges (for Mkhedruli Georgian) U+10D0 (ა) GEORGIAN LETTER AN ..U+10FA (ჺ) GEORGIAN LETTER AIN *and* U+10FD (ჽ) GEORGIAN LETTER AEN ..U+10FF (ჿ) GEORGIAN LETTER LABIAL SIGN
Upper	General_Category = Titlecase_Letter, *or* Uppercase = Yes *and* not in the ranges (for Mtavruli Georgian) U+1C90 (Ა) GEORGIAN MTAVRULI CAPITAL LETTER AN ..U+1CBA (Ჺ) GEORGIAN MTAVRULI CAPITAL LETTER AIN *and* U+1CBD (Ჽ) GEORGIAN MTAVRULI CAPITAL LETTER AEN ..U+1CBF (Ჿ) GEORGIAN LETTER MTAVRULI CAPITAL LABIAL SIGN
OLetter	Alphabetic = Yes, *or* U+00A0 NO-BREAK SPACE (NBSP), *or* U+05F3 (׳) HEBREW PUNCTUATION GERESH *and* Lower = No

	and Upper = No *and* Sentence_Break ≠ Extend
Numeric	Line_Break = Numeric
ATerm	U+002E (.) FULL STOP U+2024 (․) ONE DOT LEADER U+FE52 (﹒) SMALL FULL STOP U+FF0E (．) FULLWIDTH FULL STOP
SContinue	U+002C (,) COMMA U+002D (-) HYPHEN-MINUS U+003A (:) COLON U+003B (;) SEMICOLON U+037E (;) GREEK QUESTION MARK U+055D (՝) ARMENIAN COMMA U+060C (،) ARABIC COMMA U+060D (؍) ARABIC DATE SEPARATOR U+07F8 (߸) NKO COMMA U+1802 (᠂) MONGOLIAN COMMA U+1808 (᠈) MONGOLIAN MANCHU COMMA U+2013 (–) EN DASH U+2014 (—) EM DASH U+3001 (、) IDEOGRAPHIC COMMA U+FE10 (︐) PRESENTATION FORM FOR VERTICAL COMMA U+FE11 (︑) PRESENTATION FORM FOR VERTICAL IDEOGRAPHIC COMMA U+FE13 (︓) PRESENTATION FORM FOR VERTICAL COLON U+FE14 (︔) PRESENTATION FORM FOR VERTICAL SEMICOLON U+FE31 (︱) PRESENTATION FORM FOR VERTICAL EM DASH U+FE32 (︲) PRESENTATION FORM FOR VERTICAL EN DASH U+FE50 (﹐) SMALL COMMA U+FE51 (﹑) SMALL IDEOGRAPHIC COMMA U+FE54 (﹔) SMALL SEMICOLON U+FE55 (﹕) SMALL COLON U+FE58 (﹘) SMALL EM DASH U+FE63 (﹣) SMALL HYPHEN-MINUS U+FF0C (，) FULLWIDTH COMMA U+FF0D (－) FULLWIDTH HYPHEN-MINUS U+FF1A (：) FULLWIDTH COLON U+FF1B (；) FULLWIDTH SEMICOLON U+FF64 (､) HALFWIDTH IDEOGRAPHIC COMMA
STerm	Sentence_Terminal = Yes *and not* ATerm

Close	General_Category = Open_Punctuation, *or*
	General_Category = Close_Punctuation, *or*
	Line_Break = Quotation
	and not U+05F3 (׳) HEBREW PUNCTUATION GERESH
	and ATerm = No
	and STerm = No

Any	*This is not a property value; it is used in the rules to represent any code point.*

5.1.1 Sentence Boundary Rules

The table of sentence boundary rules uses the macro values listed in Table 4a. Each macro represents a repeated union of the basic Sentence_Break property values and is shown in boldface to distinguish it from the basic property values.

Table 4a. Sentence_Break Rule Macros

Macro	Represents
ParaSep	(Sep | CR | LF)
SATerm	(STerm | ATerm)

Break at the start and end of text, unless the text is empty.			
SB1	sot	÷	Any
SB2	Any	÷	eot
Do not break within CRLF.			
SB3	CR	×	LF
Break after paragraph separators.			
SB4	**ParaSep**	÷	
*Ignore Format and Extend characters, except after sot, **ParaSep**, and within CRLF. (See Section 6.2, Replacing Ignore Rules.) This also has the effect of: Any × (Format | Extend)*			
SB5	X (Extend | Format)*	→	X
Do not break after full stop in certain contexts. [See note below.]			
SB6	ATerm	×	Numeric
SB7	(Upper | Lower) ATerm	×	Upper
SB8	ATerm Close* Sp*	×	(¬(OLetter | Upper | Lower | **ParaSep** | **SATerm**))* Lower
SB8a	**SATerm** Close* Sp*	×	(SContinue | **SATerm**)
Break after sentence terminators, but include closing punctuation, trailing spaces, and any paragraph separator. [See note below.]			
SB9	**SATerm** Close*	×	(Close | Sp | **ParaSep**)
SB10	**SATerm** Close* Sp*	×	(Sp | **ParaSep**)
SB11	**SATerm** Close* Sp* **ParaSep**?	÷	
Otherwise, do not break.			
SB998	Any	×	Any

Notes:

- Rules SB6–SB8 are designed to forbid breaks after ambiguous terminators (primarily U+002E FULL STOP) within strings such as those shown in *Figure 3*. The contexts which forbid breaks include occurrence directly before a number, between uppercase letters, when followed by a lowercase letter (optionally after certain punctuation), or when followed by certain continuation punctuation such as a comma, colon, or semicolon. These rules permit breaks in strings such as those shown in *Figure 4*. They cannot detect cases such as “...Mr. Jones...”; more sophisticated tailoring would be required to detect such cases.
- Rules SB9–SB11 are designed to allow breaks after sequences of the following form, but not within them:
    - (STerm | ATerm) Close* Sp* (Sep | CR | LF)?
- Note that in unusual cases, a word segment (determined according to *Section 4 Word Boundaries*) may span a sentence break (according to *Section 5 Sentence Boundaries*). Inconsistencies between word and sentence boundaries can be reduced by customizing SB11 to take account of whether a period is followed by a character from a script that does not normally require spaces between words.
- Users can run experiments in an interactive online demo to observe default word and sentence boundaries in a given piece of text.

Figure 3. Forbidden Breaks on “.”

c.	d
3.	4
U.	S.
... the resp.	leaders are ...
... etc.)’	‘(the ...

Figure 4. Allowed Breaks on “.”

She said “See spot run.”	John shook his head. ...
... etc.	它们指...
...理数字.	它们指...

6 Implementation Notes

6.1 Normalization

The boundary specifications are stated in terms of text normalized according to Normalization Form NFD (see Unicode Standard Annex #15, “Unicode Normalization Forms” [UAX15]). In practice, normalization of the input is not required. To ensure that the same results are returned for canonically equivalent text (that is, the same boundary positions will be found, although those may be represented by different offsets), the grapheme cluster boundary specification has the following features:

- There is never a break within a sequence of nonspacing marks.
- There is never a break between a base character and subsequent nonspacing marks.

The specification also avoids certain problems by explicitly assigning the Extend property value to certain characters, such as U+09BE (◌া) BENGALI VOWEL SIGN AA, to deal with particular compositions.

The other default boundary specifications never break within grapheme clusters, and they always use a consistent property value for each grapheme cluster as a whole.

6.2 Replacing Ignore Rules

An important rule for the default word and sentence specifications ignores Extend and Format characters. The main purpose of this rule is to always treat a grapheme cluster as a single character—that is, to not break a single grapheme cluster across two higher-level segments. For example, both word and sentence specifications do not distinguish between L, V, T, LV, and LVT: thus it does not matter whether there is a sequence of these or a single one. Format characters are also ignored by default, because these characters are normally irrelevant to such boundaries.

The “Ignore” rule is then equivalent to making the following changes in the rules:

Replace the “Ignore” rule by the following, to disallow breaks within sequences (except after CRLF and related characters):		
Original		**Modified**
X (Extend | Format)*→X	⇒	(¬Sep) × (Extend | Format)
In all subsequent rules, insert (Extend | Format) after every boundary property value, except in negations (such as ¬(OLetter | Upper ...). (It is not necessary to do this after the final property, on the right side of the break symbol.) For example:*		
Original		**Modified**
X Y × Z W	⇒	X (Extend | Format)* Y (Extend | Format)* × Z (Extend | Format)* W
X Y ×	⇒	X (Extend | Format)* Y (Extend | Format)* ×
An alternate expression that resolves to a single character is treated as a whole. For example:		
Original		**Modified**
(STerm | ATerm)	⇒	(STerm | ATerm) (Extend | Format)*
*This is **not** interpreted as:*		
	⇏	(STerm (Extend | Format)* | ATerm (Extend | Format)*)

> **Note:** Where the “Ignore” rule uses a different set, such as (Extend | Format | ZWJ) instead of (Extend | Format), the corresponding changes would be made in the above replacements.

The “Ignore” rules should not be overridden by tailorings, with the possible exception of remapping some of the Format characters to other classes.

6.3 State Machines

The rules for grapheme clusters can be easily converted into a regular expression, as in *Table 1b, Combining Character Sequences and Grapheme Clusters*. It must be evaluated starting at a known boundary (such as the start of the text), and it will determine the next boundary position. The resulting regular expression can also be used to generate fast, deterministic finite-state machines that will recognize all the same boundaries that the rules do.

The conversion into a regular expression is very straightforward for grapheme cluster boundaries. It is not as easy to convert the word and sentence boundaries, nor the more complex line boundaries [UAX14]. However, it is possible to also convert their rules into fast, deterministic finite-state machines that will recognize all the same boundaries that the rules do. The implementation of text segmentation in the ICU library follows that strategy.

For more information on Unicode Regular Expressions, see Unicode Technical Standard #18, “Unicode Regular Expressions” [UTS18].

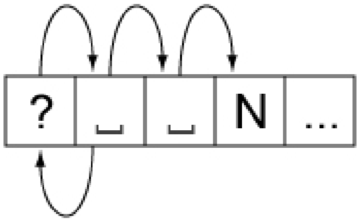
6.4 Random Access

Random access introduces a further complication. When iterating through a string from beginning to end, a regular expression or state machine works well. From each boundary to find the next boundary is very fast. By constructing a state table for the reverse direction from the same specification of the rules, reverse iteration is possible.

However, suppose that the user wants to iterate starting at a random point in the text, or detect whether a random point in the text is a boundary. If the starting point does not provide enough context to allow the correct set of rules to be applied, then one could fail to find a valid boundary point. For example, suppose a user clicked after the first space after the question mark in “Are␣you␣there?␣ ␣No,␣I’m␣not”. On a forward iteration searching for a sentence boundary, one would fail to find the boundary before the “N”, because the “?” had not been seen yet.

A second set of rules to determine a “safe” starting point provides a solution. Iterate backward with this second set of rules until a safe starting point is located, then iterate forward from there. Iterate forward to find boundaries that were located between the safe point and the starting point; discard these. The desired boundary is the first one that is not less than the starting point. The safe rules must be designed so that they function correctly no matter what the starting point is, so they have to be conservative in terms of finding boundaries, and only find those boundaries that can be determined by a small context (a few neighboring characters).

Figure 5. Random Access



This process would represent a significant performance cost if it had to be performed on every search. However, this functionality can be wrapped up in an iterator object, which preserves the information regarding whether it currently is at a valid boundary point. Only if it is reset to an arbitrary location in the text is this extra backup processing performed. The iterator may even cache local values that it has already traversed.

6.5 Tailoring

Rule-based implementation can also be combined with a code-based or table-based tailoring mechanism. For typical state machine implementations, for example, a Unicode character is typically passed to a mapping table that maps characters to boundary property values. This mapping can use an efficient mechanism such as a trie. Once a boundary property value is produced, it is passed to the state machine.

The simplest customization is to adjust the values coming out of the character mapping table. For example, to mark the appropriate quotation marks for a given language as having the sentence boundary property value Close, artificial property values can be introduced for different quotation marks. A table can be applied after the main mapping table to map those artificial character property values to the real ones. To change languages, a different small table is substituted. The only real cost is then an extra array lookup.

For code-based tailoring a different special range of property values can be added. The state machine is set up so that any special property value causes the state machine to halt and return a particular exception value. When this exception value is detected, the higher-level process can call specialized code according to whatever the exceptional value is. This can all be encapsulated so that it is transparent to the caller.

For example, Thai characters can be mapped to a special property value. When the state machine halts for one of these values, then a Thai word break implementation is invoked internally, to produce boundaries within the subsequent string of Thai characters. These boundaries can then be cached so that subsequent calls for next or previous boundaries merely return the cached values. Similarly Lao characters can be mapped to a different special property value, causing a different implementation to be invoked.

7 Testing

There is no requirement that Unicode-conformant implementations implement these default boundaries. As with the other default specifications, implementations are also free to override (tailor) the results to meet the requirements of different environments or particular languages. For those who do implement the default boundaries as specified in this annex, and wish to check that that their implementation matches that specification, three test files have been made available in [Tests29].

These tests cannot be exhaustive, because of the large number of possible combinations; but they do provide samples that test all pairs of property values, using a representative character for each value, plus certain other sequences.

A sample HTML file is also available for each that shows various combinations in chart form, in [Charts29]. The header cells of the chart show the property value. The body cells in the chart show the *break status*: whether a break occurs between the row property value and the column property value. If the browser supports tool-tips, then hovering the mouse over a header cell will show a sample character, plus its abbreviated general category and script. Hovering over the break status will display the number of the rule responsible for that status.

> **Note:** Testing two adjacent characters is insufficient for determining a boundary.

The chart may be followed by some test cases. These test cases consist of various strings with the break status between each pair of characters shown by blue lines for breaks and by whitespace for non-breaks. Hovering over each character (with tool-tips enabled) shows the character name and property value; hovering over the break status shows the number of the rule responsible for that status.

Due to the way they have been mechanically processed for generation, the test rules do not match the rules in this annex precisely. In particular:

1. The rules are cast into a more regex-style.

2. The rules “sot ÷”, “÷ eot”, and “÷ Any” are added mechanically and have artificial numbers.
3. The rules are given decimal numbers without prefix, so rules such as WB13a are given a number using tenths, such as 13.1.
4. Where a rule has multiple parts (lines), each one is numbered using hundredths, such as
   - 21.01) × $BA
   - 21.02) × $HY
   - ...
5. Any “treat as” or “ignore” rules are handled as discussed in this annex, and thus reflected in a transformation of the rules not visible in the tests.

The mapping from the rule numbering in this annex to the numbering for the test rules is summarized in *Table 5*.

Table 5. Numbering of Rules

Rule in This Annex	Test Rule	Comment
xx1	0.2	sot (start of text)
xx2	0.3	eot (end of text)
SB8a	8.1	Letter style
WB13a	13.1	
WB13b	13.2	
GB999	999.0	Any
WB999		

Note: Rule numbers may change between versions of this annex.

8 Hangul Syllable Boundary Determination

In rendering, a sequence of jamos is displayed as a series of syllable blocks. The following rules specify how to divide up an arbitrary sequence of jamos (including nonstandard sequences) into these syllable blocks. The symbols L, V, T, LV, LVT represent the corresponding Hangul_Syllable_Type property values; the symbol M for combining marks.

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 6*. In all cases other than those shown in *Table 6*, a syllable break occurs before and after any jamo or precomposed Hangul syllable. As for other characters, any combining mark between two conjoining jamos prevents the jamos from forming a syllable block.

Table 6. Hangul Syllable No-Break Rules

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

		L × LV L × LVT
V or LV	V or T	V × V V × T LV × V LV × T
T or LVT	T	T × T LVT × T
Jamo, LV or LVT	Combining marks	L × M V × M T × M LV × M LVT × M

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

For information on the behavior of Hangul compatibility jamos in syllables, see *Section 18.6, Hangul* of [Unicode].

8.1 Standard Korean Syllables

- *Standard Korean syllable block:* A sequence of one or more L followed by a sequence of one or more V and a sequence of zero or more T, or any other sequence that is canonically equivalent.

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

Using regular expression notation, a canonically decomposed standard Korean syllable block is of the following form:

L+ V+ T*

Arbitrary standard Korean syllable blocks have a somewhat more complex form because they include any canonically equivalent sequence, thus including precomposed Korean syllables. The regular expressions for them have the following form:

(L+ V+ T*) | (L* LV V* T*) | (L* LVT T*)

All standard Korean syllable blocks used in modern Korean are of the form <L V T> or <L V> and have equivalent, single-character precomposed forms.

Old Korean characters are represented by a series of conjoining jamos. While the Unicode Standard allows for two L, V, or T characters as part of a syllable, KS X 1026-1 only allows single instances. Implementations that need to conform to KS X 1026-1 can tailor the default rules in *Section 3.1 Default Grapheme Cluster Boundary Specification* accordingly.

8.2 Transforming into Standard Korean Syllables

A sequence of jamos that do not all match the regular expression for a standard Korean syllable block can be transformed into a sequence of standard Korean syllable blocks by the correct insertion of choseong fillers (L$_f$) and jungseong fillers (V$_f$). This transformation of a string of text into standard Korean syllables is performed by determining the syllable breaks as explained in the earlier subsection “Hangul Syllable Boundaries,” then inserting one or two fillers as necessary to transform each syllable into a standard Korean syllable as shown in *Figure 6*.

Figure 6. Inserting Fillers

L [^V] → L V$_f$ [^V]
[^L] V → [^L] L$_f$ V
[^V] T → [^V] L$_f$ V$_f$ T

In *Figure 6*, [^X] indicates a character that is not X, or the absence of a character.

In *Table 7*, the first row shows syllable breaks in a standard sequence, the second row shows syllable breaks in a nonstandard sequence, and the third row shows how the sequence in the second row could be transformed into standard form by inserting fillers into each syllable. Syllable breaks are shown by *middle dots* “·”.

Table 7. Korean 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 · LLVV
3	LLTTVVTTVVLLVV	→	LLV$_f$ · L$_f$ V$_f$ TT · L$_f$ VVTT · L$_f$ VV · LLVV

Acknowledgments

Mark Davis is the author of the initial version and has added to and maintained the text of this annex through Version 14.0. Laurențiu Iancu assisted in updating it for Versions 7.0 through 10.0.

Thanks to Julie Allen, Asmus Freytag, Manish Goregaokar, Andy Heninger, Ted Hopp, Tsuyoshi Ito, Martin Hosken, Michael Kaplan, Johan Curcio Lindström, Eric Mader, Otto Stolz, Steve Tolkin, Ken Whistler, and Karl Williamson for their feedback on this annex, including earlier versions.

References

For references for this annex, see Unicode Standard Annex #41, “Common References for Unicode Standard Annexes.”

Modifications

The following summarizes modifications from the previous published version of this annex.

Revision 44

- **Proposed Update** for Unicode 16.0.0.

- Table 2, Grapheme_Cluster_Break Property Values: Updated the definition of GCB=V to include Kirat Rai vowel signs.
- Section 3.1.1, Grapheme Cluster Boundary Rules: Updated the description of rules GB6–GB8 and added a note to account for the extension of conjoining behaviour beyond Hangul Jamo.
- In Section 3, clarified that legacy grapheme clusters are actively maintained.
- Updated the definition of SB=STerm to subtract ATerm, to ensure that they are disjoint.
- Table 4, Sentence_Break Property Values: Updated the definition of SContinue to include several Semicolon & related characters.
- Section 4, Word Boundaries: Updated introduction with improved wording.
- Clarify "user-perceived characters" and explain how grapheme clusters are an attempt to approximate them.

~~Revision 43~~

- ~~**Reissued** for Unicode 15.1.0.~~
- ~~GCB Table 1c, Regex Definitions~~
    - ~~Changed definition of `crlf`~~
    - ~~Updated the definition of the conjunctCluster regular expression~~
    - ~~Updated regex definitions for consistency of use of the term "RI-sequence"~~
- ~~Table 3, Word_Break Property Values~~
    - ~~Excluded GCB = Prepend from Format~~
    - ~~Added U+070F to ALetter~~
    - ~~Changed the derivation of WB=Numeric to avoid unintended changes as the Line_Break property gets refined; the fullwidth digits remain WB=Numeric because of their General_Category~~
- ~~Added conformance rules for each type of segmentation, in Section 2 Conformance~~
- ~~Removed note in Section 3 about boundaries.~~
- ~~Updated wording about aksaras in Section 3.~~
- ~~Added note that each emoji sequence is a single grapheme cluster to 3.1.1 Grapheme Cluster Boundary Rules~~
- ~~Updated rule GB9c to use the Indic_Conjunct_Break property instead of macros.~~
- ~~Updated paragraph below Figure 2 to indicate that the relationship of line break/word break boundaries is script-specific.~~
- ~~Changed note regarding boundaries in Section 7, Testing~~

Modifications for previous versions are listed in those respective versions.